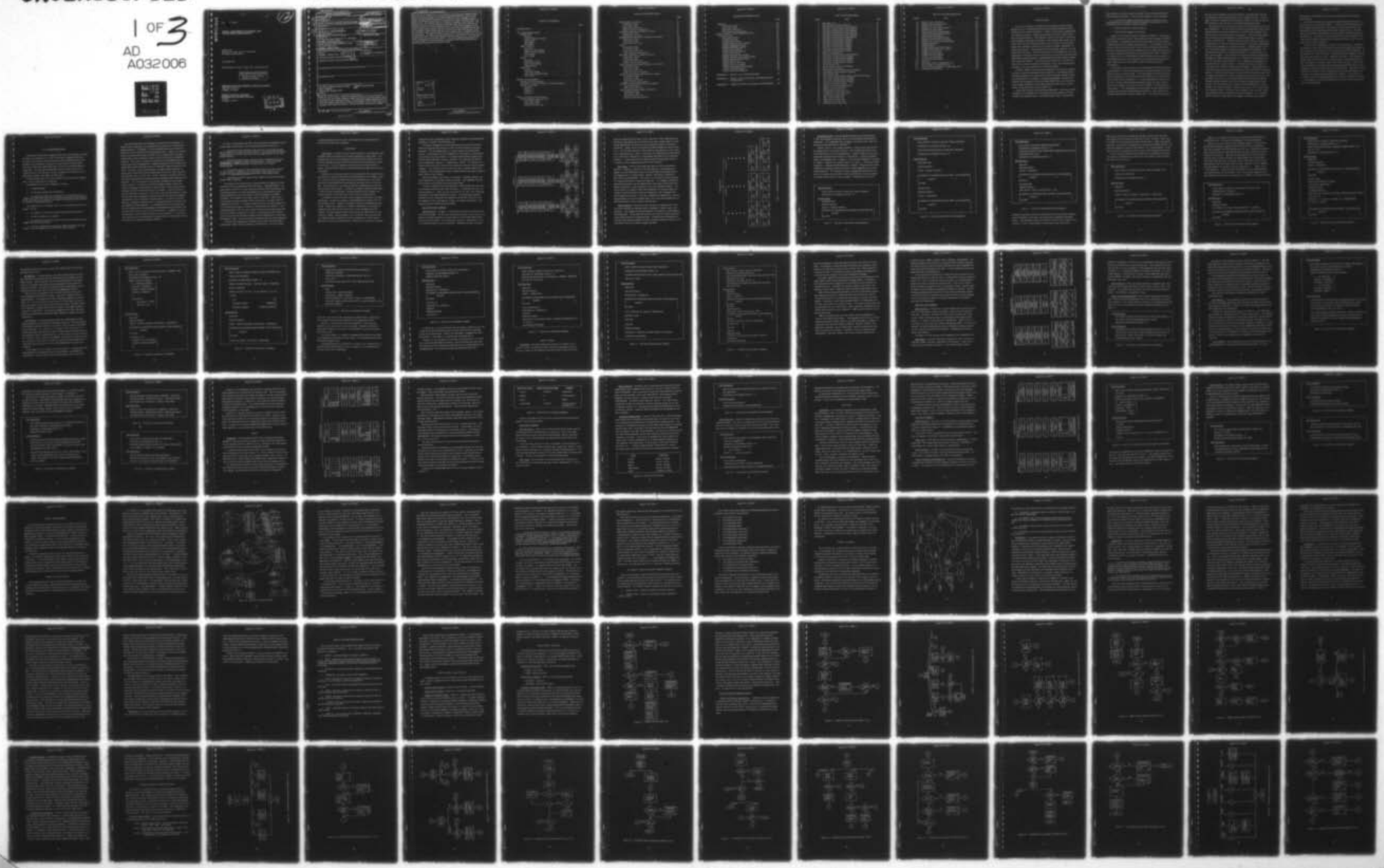
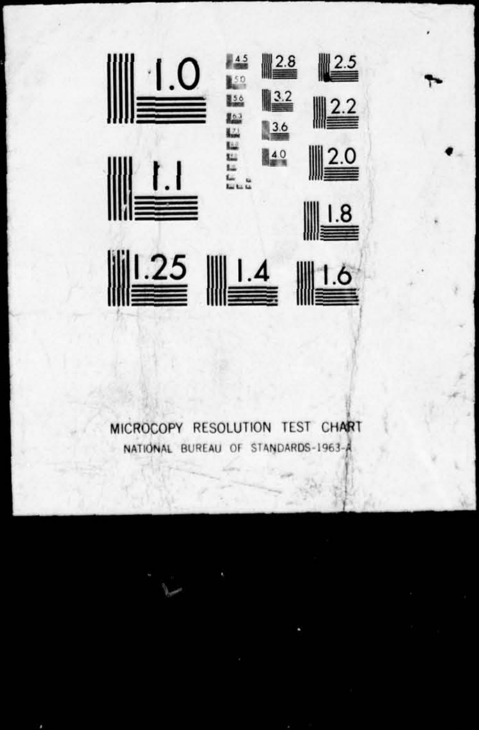
AD-A032 006   LOGICON INC SAN DIEGO CALIF   F/G 9/2
ADAPT I PRELIMINARY FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION.(U)
NOV 76 L R ERICKSON, M E SOLEGLAD   N00014-76-C-0899
UNCLASSIFIED   76-C-0899-2   NL

1 OF 3
AD
A032006




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Report 76-C-0899-2

(12)

Rep. 76-C-0899-2

ADAPT I PRELIMINARY FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION

Logicon, Inc.
4010 Sorrento Valley Blvd. P.O. Box 80158
San Diego, California 92138

1 November 1976

Interim Report for Period 1 August 1976 — 1 November 1976

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
1400 Wilson Boulevard
Arlington, VA 22209

OFFICE OF NAVAL RESEARCH
INFORMATION SYSTEMS PROGRAM
Code 437
Arlington, VA 22217

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
14 1. REPORT NUMBER 76-C-0899-2	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) 6 ADAPT I Preliminary Functional and System Design Specification.		5. TYPE OF REPORT & PERIOD COVERED Interim Report ~~1 Aug 1976 - 1 Nov 1976~~ 1 Aug - 1 Nov 76
		6. PERFORMING ORG. REPORT NUMBER 76-C-0899-2
7. AUTHOR(s) 10 L. R. Erickson, M. E. Soleglad and S. L. Westermark		8. CONTRACT OR GRANT NUMBER(s) 15 N00014-76-C-0899
9. PERFORMING ORGANIZATION NAME AND ADDRESS Logicon, Inc. 4010 Sorrento Valley Boulevard, P.O. Box 80158 San Diego, California 92138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209	11	12. REPORT DATE 1 November 1976
		13. NUMBER OF PAGES 214
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	12 219p.	15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data Base Management Systems (DBMS)	Language Transformations
Query Languages	UNIX
Uniform Data Language (UDL)	
Network	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Under the ADAPT project, a prototype intelligent terminal will be developed which provides users and/or other systems a uniform interface for accessing multiple online data bases located on different systems. The underlying technology applied by ADAPT will be the transformation from one uniform data language, UDL, to other target query languages which reside on a network. The four data base systems/query languages

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

406012

that UDL will be transformed to are the SIGINT On-Line Information System (SOLIS), Defense Intelligence Agency On-Line System (DIAOLS)/Intelligence Support System (ISS), Data Base Management System 1100 (DMS-1100)/Query Language Processor (QLP), and Technical Information Processing Systems (TIPS)/TIPS Interrogation Language (TILE). The ADAPT system is comprised of 11 distinct UNIX processes. Significant processes are the Data Definition Language (DDL) Processor which provides ADAPT I users with a full file definition facility via an interactive language; two language transformation processes (one for true interactive query languages and the other for transaction-oriented (batch) data base systems); and a data base response translation processer which maps all distant host data responses onto a compatible UDL data representation. All ADAPT I processes operate over a UNIX hierarchical file structure composed of a set of UNIX directories and internally structured files. The structured files comprise the global data for ADAPT I. Key to the language transformations performed by ADAPT I is the normalization of selection-criteria (generalized boolean expressions) to disjunctive-normal-form (DNF).

1

2

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Cont)

INTRODUCTION

This report constitutes a preliminary functional and system design specification for the ARPA Data Base Access and Presentation Terminal (ADAPT) system, phase I (ADAPT I). ADAPT, which will be implemented in a sequence of phases, will provide network users with a uniform data base accessing interface. This interface will be in the form of a data base query language comprised of two parts: a set of logical data structures in which a user can view all network files, and a set of user-oriented statements and commands by which users can query, display, manipulate, or update network files. These statements and commands "operate" over the logical data structures. ADAPT also provides to privileged users two sub-languages for defining network-resident files in terms of the aforementioned logical data structures and for specifying distant host file-specific information required by ADAPT. The primary goal of ADAPT is to provide network users a uniform interface for data base access, thus eliminating the learning curve associated with each dissimilar system.

Initially, ADAPT will only provide general file interrogation capabilities plus a limited data display facility (ADAPT I). In additional phases, full file maintenance and sophisticated report generation facilities will be provided (ADAPT II), complete "programming language" constructs for dynamic file manipulation will be provided (ADAPT III), and a local data base manager capability will be added which will allow users to create, query, and maintain files locally (ADAPT IV).

While this document was undergoing preparation, some major system interfaces to be supplied in the Technology Transfer Research Facility (TTRF) were going through a "conceptual design" phase. Consequently, these interfaces and their impact on ADAPT I have not been addressed in this preliminary document. These system interfaces are the following:

a. User/terminal authentication with respect to overall security, concerning entry into ADAPT I and the individual authentication of user/terminal accesses to specific distant host files.

b. Uniform distant host logon sequences.

c. TTRF network logging requirements.

Another TTRF system interface required by ADAPT I is "batch query processing" (also known as the "Batch Network Control Program" (BNCP)). This interface is addressed briefly in this document in order to complete certain narratives describing system data flow. It is important to note here, however, that at the time of this writing, the four TTRF interfaces have been well identified by TTRF personnel and are currently being considered as actual centralized UNIX processes operating in the TTRF. Therefore, it is believed that, if the definitions of these interfaces are provided in a reasonable time frame, the incorporation of the appropriate interfaces to these TTRF processes into ADAPT I will not cause any schedule perturbations.

Also missing from this report is the description of the file transformation dictionaries. These file dictionaries were going through their design phase at the time of this writing. Their descriptions (as well as the interface descriptions to TTRF processes just discussed) will be provided in a final version of this specification. It is also appropriate to mention here that the Transformation Definition Language (TDL) for target system files will be supplied in the final version of "ADAPT I Uniform Data Language (UDL)." This sublanguage provides the file-specific information required to generate file transformation dictionaries. Each network file utilized under ADAPT I must be defined twice. The UDL interpretation of the file must be supplied via the Data Definition Language (DDL) and the transformation-specific information must be supplied via TDL. Interestingly, a local file capability as envisioned for ADAPT IV requires only the DDL definition, since the file is created and manipulated locally.

The remainder of this document consists of four additional sections and three appendices. The next section provides a functional description of the transformations required to transform the Uniform Data Language (UDL) into four distant host data base query languages. These data base management systems and query languages are DIAOLS/ISS, DMS-1100/QLP, SOLIS, and TIPS/TILE. This section gives strong evidence to support the statement that UDL can be transformed to these four data base management system query languages in a consistent manner. Although ADAPT I deals only with transformations involving file interrogations and display, the complete transformation set has been provided; i. e. , interrogation, display, and file update. Since ADAPT II will provide additional capabilities, including file update transformations, it is important to demonstrate that the UDL subset provided in ADAPT I can be expanded (not changed) to incorporate the new transformations. The third section gives a general description of the ADAPT I operational environment. This includes a detailed description of the file environment over which the ADAPT I processes operate. All UNIX file path names are described and the mechanisms and/or naming conventions for constructing dynamic path names are given. Also included in this section is a brief description of the interactive ADAPT I System Generation program that will be required to generate the initial ADAPT I file environment and to register and/or delete ADAPT I users. Finally, this section provides a general description of the overall data flow between ADAPT I processes with respect to UDL and DDL statement/command entries by users and network responses. It is important that this section be read before one attempts to read the more detailed descriptions of the ADAPT I processes and global data files provided in later sections.

The fourth section provides somewhat detailed descriptions of the 11 processes now envisioned for ADAPT I. Included in these descriptions are lists of global data utilization, descriptions of important functions, and general flow charts illustrating the data flow of each process. The fifth section provides detailed descriptions of the ADAPT I global data functions and data files. The ADAPT I file environment discussion provided in the third section should be studied before reading this section, since it ties all global data structures together.

There are three appendices included in this specification. Appendix A provides the actual YACC (Yet Another Compiler-Compiler) input specifications (grammer descriptions) for the ADAPT I UDL and DDL languages. In the final version of this document, this appendix will be expanded to include the YACC specification for TDL. Appendix B provides a very detailed example of the interaction between a file dictionary for a hypothetical UDL file and sample record data as they would exist in ADAPT I. Before reading this appendix, one should study the applicable global data structure descriptions in the fifth section of the basic specification. Appendix C contains a detailed description of the Boolean Expression Normalizer (BEN) function that is currently under investigation for ADAPT I. This function is crucial to many of the transformations that are being performed by ADAPT I. This appendix is somewhat independent of the rest of this document and can be read separately.

UDL TRANSFORMATIONS

The ADAPT Uniform Data Language (UDL) has been designed to present a uniform user interface to a variety of data base management systems. The users of these data base management systems represent many areas of interest and, therefore, represent a varied range of requests which must all be adequately satisfied through UDL and its associated software components. The data base management systems (referred to here as "target systems") provide their own query languages to enable the user to interrogate, display, and update the data bases.

The initial objectives to be met by ADAPT and its respective language, UDL, can be divided into two categories.

In designing UDL as a pure language, it must be:

a. A single language.

b. Functionally complete and independent.

c. An evolutionary language with respect to its functional power; i.e., simple and complex versions of the same function should be representable by the same basic root command, the difference being only in the presence or absence of optional parameters.

In designing UDL as a language which can be translated into other languages, it must:

a. Be capable of accessing existing and potential target systems.

b. Present a uniform user interface.

c. Be constructed such that its data structures and statements can be mapped onto the target systems.

d. Provide a consistent interpretation of data structures and statements as they are applied across the various target systems.

The transformation of one language into many diverse languages is a difficult task, especially if this transformation sequence is to satisfy foregoing condition d. Ostensibly, it seems possible that many or most UDL functions could be satisfied by transformations alone. That is, the function itself is not processed locally in ADAPT, but is transformed to an appropriate target system statement sequence where the function is performed remotely. This notion is caused in most part by the assumption that all target system query languages are complete enough that each contains some hypothetical minimal function set that is sufficient to generate all UDL functions. Therefore, with a little ingenuity, a target system statement sequence can be found which can be mapped onto by any given UDL statement. Unfortunately, this is not the case. Great variability is present in the target system languages where many languages completely lack important UDL functions. This is particularly true with data manipulation and complex display functions. Therefore, the ADAPT design strategy can not rely solely on a language transformation technique and, in fact, this particular technique should be minimized as much as possible. If this is not done, it will be impossible to construct a UDL that is both powerful enough to satisfy the basic data base management requirements and one that can be applied consistently across a multitude of target systems. In order to meet this objective, language transformations are being performed in two areas only: interrogation and file updating. It is mandatory that these two functional areas be implemented through language transformations since ADAPT does not have direct access to the various network files. In addition, a limited transformation of UDL display functions is also required; i. e., those functions which cause the transmission of target system file information through the network for subsequent display in ADAPT.

In order to demonstrate that UDL can be interpreted consistently across various target system files, four transformation goals must be met:

a. Demonstrate that each major data structure of all target systems can be mapped onto a legal UDL data structure. Inconsistent interpretations of data structures based on a particular file or target system are not valid transformations.

b. Demonstrate that all data contained in these mapped files can be interrogated and updated via legal UDL statements. Using inconsistent interpretations of UDL statements to achieve this is not a valid transformation.

c. Demonstrate that the UDL statements which were used to satisfy the interrogation or updating of files for a given target system can be mapped onto functionally equivalent statements of that target system.

d. Demonstrate that all required UDL outputs are achievable through target system responses.

The material which follows gives strong evidence to support the statement that UDL does indeed meet these four goals. Although ADAPT I, the initial phase of ADAPT, is only concerned with data base interrogation and limited data display, it seems appropriate here to demonstrate the full transformability of UDL with respect to both interrogation and update. Therefore, the examples which follow utilize a full UDL, a UDL which is much larger and more complete than the one which is currently under development for ADAPT I. It might be added here that, upon the existence of a local data base manager in ADAPT as recommended for ADAPT IV, the transformability of UDL into other target systems becomes more attainable. Based on research conducted by Logicon, it was discovered that reliance upon a local data base manager strengthened and expanded the overall transformability of UDL, even to somewhat unorthodox data base query languages.

Four data base management systems are discussed in the following: DIAOLS/ISS, DMS-1100/QLP, SOLIS, and TIPS/TILE. For each system, a simple hypothetical file is used to demonstrate the transformability of UDL to the various query languages.

DIAOLS/ISS

OVERVIEW – The DIA On-LIne System (DIAOLS) is an intellegence data handling system developed by the Defense Intelligence Agency (DIA). The primary reason for the development of DIAOLS in the late 1960s was to provide the intelligence analyst with direct access to the information he needs to do his job. As such, DIAOLS provides for user servicing in three modes: interactive time sharing, remote batch processing, and local batch processing.

DIAOLS is currently operational on two Honeywell G-635 computers supported by the General Comprehensive Operating Supervisor (GCOS), release F8. The two G-635s are called System One and System Two, and utilize primarily Honeywell 6000 series peripherals including DSU-191 disk storage. Three Datanet 30s currently provide the communications front-end processing for both systems. Terminals currently available include Model 37 ASR and KSR teletypewriters, Raytheon DIDS 400 CRT displays with Inktronic printers, and COPE 1200 remote batch terminals (RBTs).

A modified version of GCOS is used to support processing of classified information. System One supports compartmented processing to the Top Secret/SI/SAO level. System Two supports only non-compartmented processing. System One supports two software subsystems: the Intelligence Support System (ISS) and the Community On-Line Intelligence System (COINS). The ISS supports analyst requirements for information storage, retrieval, display, and maintenance in an interactive time-sharing mode. The COINS provides analysts with access to data bases at other national agencies in a store and forward mode. System Two is basically a standard

Honeywell system supporting several language subsystems including BASIC, FORTRAN, CARDIN, TEXT EDITOR, etc.

The ISS was designed as a generalized storage and retrieval system which would quickly allow analysts to seek out direct answers to specific questions. The key design factor was rapid response to unpredictable queries. Each element of a file designated as retrievable is equally likely to be used as a key for retrieval. The kinds of data entities supported by the ISS include elements, records, and files. The types of data allowed have an alphabetic, numeric, or alphanumeric format. The logical structuring of data in the ISS is hierarchical using inverted lists to provide rapid response. The storage structure used by the ISS is random within a file. A Random Access Management System (RAMS) performs all logical/physical input/output for ISS routines.

Data interrogation, processing, maintenance, updating, display, and report generation are provided online by the ISS. Data definition and data base creation are provided via the local batch mode.

In order to illustrate the transformation of DIAOLS/ISS data structures to functionally equivalent UDL data structures, a simple file has been constructed (figure 1). Three records are shown in this file, each containing information concerning hypothetical targets. The following paragraphs examine this file showing the transformation of ISS data structures, and the mapping of UDL interrogation, display, and update statements to functionally equivalent DIAOLS/ISS statements.

DIAOLS/ISS DATA DESIGN –

Data Structures – In figure 1, all ISS data structures are represented in UDL terminology. The ISS record maps directly onto a UDL record. ISS fixed and variable elements are represented as UDL single-valued fields, such as the field USAGE. The ISS periodic element is represented in UDL as a multivalued field, such as REPORTS. Relational periodic elements,

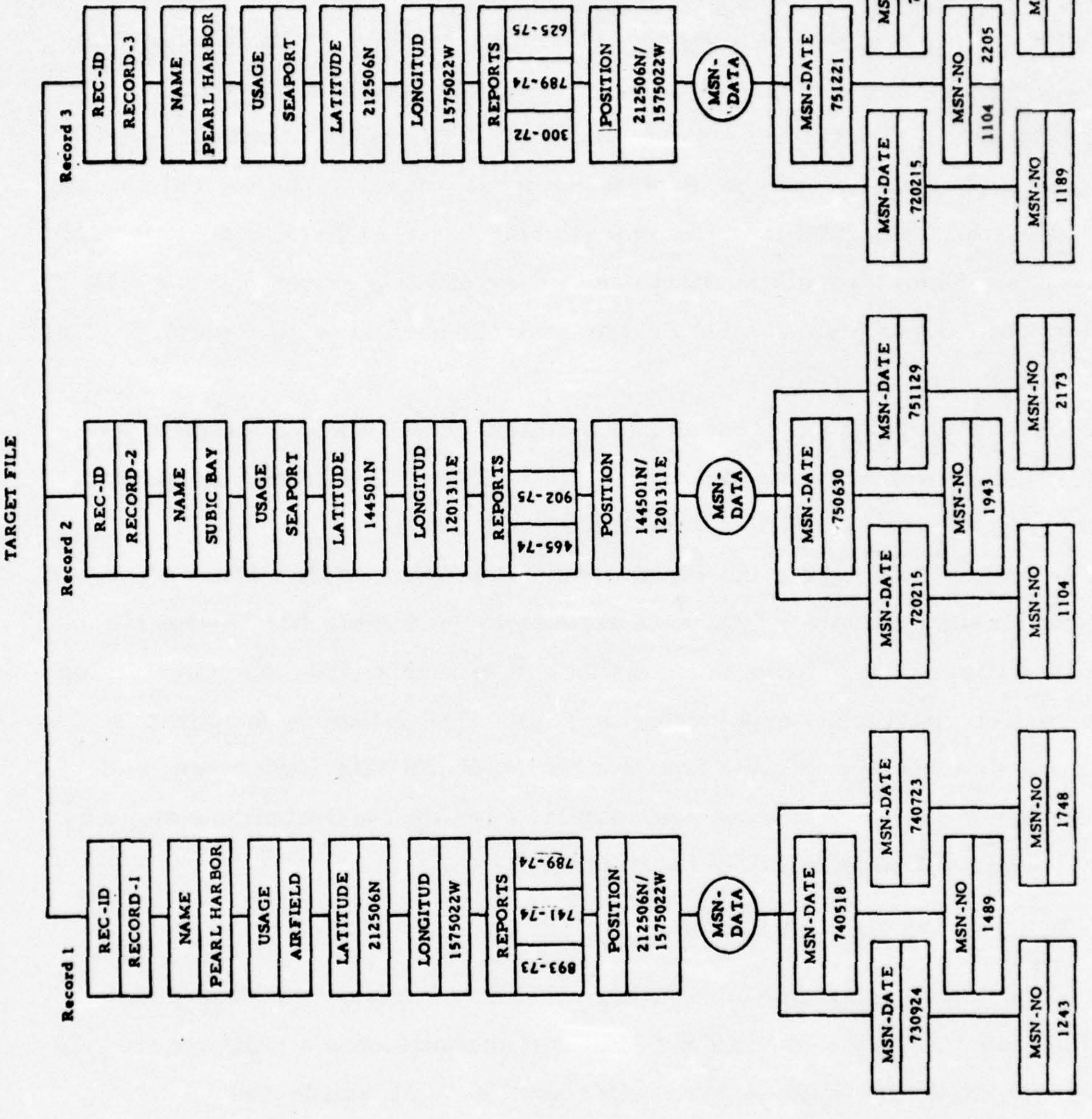


Figure 1. ISS Sample File.

the most complex data structure in ISS, maps onto a UDL repeating group. Within the repeating group, the fields may be either single-valued or multivalued depending on the actual use of the relational periodic subscript in the ISS file. Single-valued and multivalued field examples are shown in figure 1. The internal subscripts used by ISS are equivalent to an occurrence of the repeating group. Figure 2 illustrates the repeating group MSN-DATA as it would exist in DIAOLS; it is functionally equivalent to its counterpart in figure 1. Notice that all value instances with the same index form an occurrence of the repeating group MSN-DATA in figure 1.

Data Types – The five data types described for UDL are present in DIAOLS. In figure 1, only three UDL data types are represented: geographic for field POSITION; numeric for fields MSN-NO and MSN-DATE; and alphanumeric for fields NAME, USAGE, LATITUDE, LONGITUD, MSN-DATE, and REPORTS. The choice of a data type for any given field of a target system file is somewhat subjective. For example, all DIAOLS group data structures will be mapped onto a UDL single-valued field even though in DIAOLS it is the juxtaposition of several simple elements. This field will then be assigned a data type designation of variable-length text. Lengthy variable elements in DIAOLS (nongroups) will be mapped onto a fixed-length text data type. Data type mappings for alphanumeric, numeric, and geographic are more straightforward, since data type designations are determined based on their usage in DIAOLS files.

Data Attributes – Of the seven data attributes recognized under UDL, five will be utilized for ISS files: keyed, dependent, visible, major, and nonmajor. This implies that all ISS fields are searchable (to some degree) and displayable. All inverted ISS elements are treated as keyed fields and noninverted elements as dependent fields. All ISS elements designated as minimum data elements (MDEs) will have an update attribute of major. All non-MDEs will have a nonmajor update attribute.

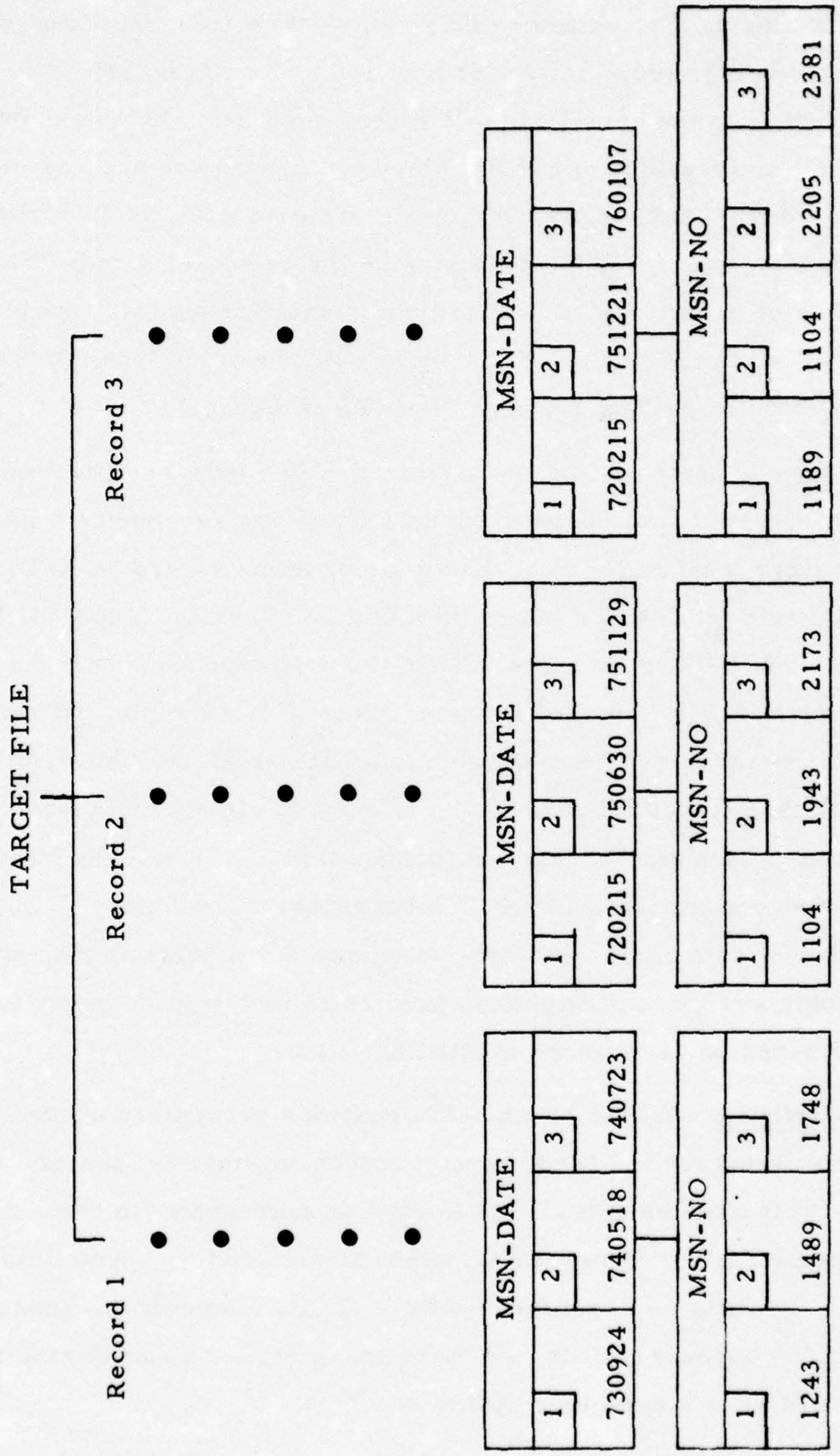


Figure 2. ISS Relational Periodic Element.

INTERROGATION – This paragraph illustrates the transformation of UDL interrogation statements onto a set of functionally equivalent ISS statements. The sample data base in figure 1 is used in conjunction with statement sequences figures 3 through 15.

Figure 3 shows a very simple example of a query which will retrieve all targets with a name of Pearl Harbor. The transformation from UDL to ISS instructions is very straightforward. Figure 4 takes this query one step further and specifies airfields at Pearl Harbor. Again the transformation is straightforward. In figure 5, the first FIND statement, identified by "label1," isolates those targets which use a seaport. The second FIND statement, a dependent FIND identified by "label2," refines the first FIND statement by applying additional selection criteria against those records already isolated. This statement is more complicated than the first one. Here, all targets with MSN-DATE 720215 and MSN-NO 1104 are isolated. It should be noted that the "scope-qualifier" of MSN-DATA forces the

UDL Statements

```
label1 FIND IN TARGET NAME EQ 'PEARL HARBOR'
[NUMBER OF RECORDS FOUND = 2]
```

ISS Statements

```
REQUEST NEW
[ENTER QUERY]
NAME = :PEARL HARBOR:.
[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
          TARGET                       2
 ACTION                                      ]
```

Figure 3. UDL/ISS Transformation (Interrogation).

UDL Statements

```
label1 FIND IN TARGET NAME EQ 'PEARL HARBOR'

[NUMBER OF RECORDS FOUND = 2]

label2 FIND SOURCE (label1) USAGE EQ 'AIRFIELD'

[NUMBER OF RECORDS FOUND = 1]
```

ISS Statements

```
REQUEST NEW

[ENTER QUERY]

NAME = :PEARL HARBOR:.

[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]

            TARGET                          2

 ACTION

REFINE NEW

[ENTER QUERY]

USAGE = :AIRFIELD:.

[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]

            TARGET                          1

 ACTION
```

Figure 4. UDL/ISS Transformation (Interrogation).

```
UDL Statements

    label1 FIND IN TARGET USAGE EQ 'SEAPORT'
    [NUMBER OF RECORDS FOUND = 2]
    label2 FIND SOURCE (label1) MSN-DATA (MSN-DATE EQ 720215
    AND MSN-NO EQ 1104)
    [NUMBER OF RECORDS FOUND = 1]

ISS Statements

    REQ NEW
    [ENTER QUERY]
    USAGE = :SEAPORT:.

    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
    [         TARGET                         2    ]
    [ACTION                                       ]

    RELATE NEW
    [ENTER QUERY]
    MSN-DATE = :720215: TO MSN-NO = :1104:.

    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
    [         TARGET                         1    ]
    [ACTION                                       ]
```

Figure 5. UDL/ISS Transformation (Interrogation).

selection to guarantee that the mission date and mission number specifications be satisfied for the same occurrence of the repeating group MSN-DATA. This statement reduces the selected record set to a single record, record 2. If the "scope-qualifier" had not been used, both record 2 and record 3 would have satisfied the selection criteria since they both have MSN-DATEs of 720215 and MSN-NOs of 1104. However, only record 2 has MSN-NO 1104 performed on MSN-DATE 720215. Again, this example provides a straightforward mapping into commands acceptable to the ISS. The RELATE statement in the ISS is equivalent to the "scope-qualifier" in UDL and is utilized in a dependent mode to operate relational periodic elements.

Search conditions against keyed fields may be combined as in figure 6. This query is functionally equivalent to the one in figure 4 and will isolate the same record, record 1.

<u>UDL Statements</u>

```
    label1 FIND IN TARGET NAME EQ 'PEARL HARBOR' AND

    USAGE EQ 'AIRFIELD'

    [NUMBER OF RECORDS FOUND = 1]
```

<u>ISS Statements</u>

```
    REQ NEW

    [ENTER QUERY]

    NAME = :PEARL HARBOR: AND USAGE = :AIRFIELD:.

    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
    [          TARGET                     1       ]
    [ACTION                                       ]
```

Figure 6. UDL/ISS Transformation (Interrogation).

Figure 7 is an example of a keyed query against the LATITUDE and LONGITUD fields. Once again the mapping is very straightforward.

Figure 8, however, poses a more complex problem. The second dependent FIND statement specifies that the circle search be performed. This may be mapped onto ISS statements in two ways. Either the ISS action GEO or CIRCLE may be requested and both will retrieve the same record (CIRCLE is used here). The only functional difference in terms of ISS is that the action CIRCLE calculates the radius distance for records lying within the circle specified. The point is that in UDL terms it must be defined ahead of time which ISS action to map to, CIRCLE or GEO. Dependent interrogations for the ISS, as shown in figures 4, 5, 7, and 8, act as "AND operations." If the two main constituents of a UDL FIND statement are connected with an "OR condition," the corresponding mapping to ISS is quite different and becomes very complicated. In reality, this will be a valid user

```
UDL Statements

    label1 FIND IN TARGET LATITUDE EQ '212506N' AND
    LONGITUD EQ '1575022W'
    [NUMBER OF RECORDS FOUND = 2]

ISS Statements

    REQ NEW
    [ENTER QUERY]
    LATITUDE = :212506N: AND LONGITUDE = :1575022W:.

    ┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
    │          TARGET                      2     │
    └ACTION                                      ┘
```

Figure 7. UDL/ISS Transformation (Interrogation).

UDL Statements

```
    label1 FIND IN TARGET USAGE EQ 'SEAPORT'
    [NUMBER OF RECORDS FOUND = 2]
    label2 FIND SOURCE (label1) POSITION INSIDE CIRCLE (100,
    %21 N/157 W)
    [NUMBER OF RECORDS FOUND = 1]
```

ISS Statements

```
    REQ NEW
    [ ENTER QUERY]
    USAGE = :SEAPORT:.
    ┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
    │          TARGET                        2     │
    └ACTION                                       ┘
    CIRCLE NEW
    [ENTER COORDINATE NAMES]
    LATITUDE, LONGITUDE
    [ENTER QUERY]
    IF INSIDE CIRCLE-1.
    [FOR CIRCLE-1 ENTER LATITUDE, LONGITUDE, AND RADIUS]
    21N, 157W, 100
    [OPTION ? L = LIST, M = MODIFY, N = RESEQUENCE,
    R = RUN, S = SAVE]
    R
    ┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
    │          TARGET                        1     │
    └ACTION                                       ┘
```

Figure 8. UDL/ISS Transformation (Interrogation).

requirement in only rare cases, and this description does not go into the procedure at this time.

ISS DISPLAY – Transformations from UDL to target systems for display operations require only that the desired fields be available from the target system. Data formatting and the various UDL print control operations are not mapped onto target system statements but are performed entirely in ADAPT. The ISS action OUTPUT will be mapped from UDL to output the desired fields which will then be manipulated in ADAPT.

Figure 9 gives an example of how to obtain the entire record which will be displayed with field name labels. This is the simplest means of displaying data. An example of the UDL formatting capability is shown in figure 10. For each record selected (record 1 in this case), the target name and position will be printed under the column titles "TARGET NAME" and "POSITION," printed left-justified at column 10 and right-justified to column 60, respectively. Two lines are skipped and the target name and position are printed left-justified and right-justified to columns 10 and 60, respectively.

ISS UPDATE – This paragraph discusses the transformation of UDL update statements to equivalent ISS statement sequences. One requirement of ISS update sequences is that the "record-id" of the record to be modified must be specified. The "record-id" is unique for each record, but it is usually functional data and not just some machine reference number (which the example in figure 1 unfortunately is). In most cases, the user will know the "record-id" of the record to be modified. Where the user does not know the record-id or where many records are to have the same modification, a keyed field may be used as the source and identification of records to be modified.

A new target record is added to the file in figure 11. Note that the UDL field POSITION is broken down into two ISS elements, LATITUDE and LONGITUDE. Also note that, since the mission date 730127 is the first

UDL Statements

```
label1 FIND IN TARGET NAME EQ 'PEARL HARBOR' AND
USAGE EQ 'AIRFIELD'
[NUMBER OF RECORDS FOUND = 1]
DISPLAY SOURCE (label1)
    [... RECORD HEADER ...
     REC-ID = RECORD-1
     NAME = PEARL HARBOR
     USAGE = AIRFIELD
     .
     .
     .
     MSN-DATA
         MSN-DATE = 730924
         MSN-NO = 1243
     .
     .
     .                    ]
```

ISS Statements

```
REQ NEW
[ENTER QUERY]
NAME = :PEARL HARBOR: AND USAGE = :AIRFIELD:.
[NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST
         TARGET                              1
 ACTION                                       ]
OUTPUT S; ALL.
    [RECORD-ID = RECORD-1
     NAME = PEARL HARBOR
     .
     .                    ]
```

Figure 9. UDL/ISS Transformation (Display).

UDL Statements

```
label1 FIND IN TARGET NAME EQ 'PEARL HARBOR' AND

USAGE EQ 'AIRFIELD'

[NUMBER OF RECORDS FOUND = 1]

DISPLAY SOURCE (label1) 'TARGET NAME', 'POSITION',

NAME, POSITION

FORMAT (PAGE, AT (10), TO (60), SKIP (2), AT (10),

TO (60))

  ┌10                                    60┐
  │TARGET NAME                       POSITION│
  └PEARL HARBOR             212506N/1575022W┘
```

ISS Statements

```
REQ NEW

[ENTER QUERY]

NAME = :PEARL HARBOR: AND USAGE = :AIRFIELD:.

┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
│        TARGET                       1       │
└ACTION                                       ┘

OUTPUT S; NAME, LATITUDE, LONGITUDE.
```

Figure 10. UDL/ISS Transformation (Display).

UDL Statements

```
CREATE IN TARGET RECORD-ID EQ 'RECORD-4',
NAME EQ 'SASEBO',
USAGE EQ 'SEAPORT', POSITION EQ %33 10 31 N/
129 43 38 E,
MSN-DATA (MSN-DATE EQ 730127, MSN-NO EQ 1199)
```

ISS Statements

```
ADD
[LISTS MDE + INSTRUCTIONS]
:RECORD-4:, NAME :SASEBO:,
USAGE :SEAPORT:, LATITUDE :331031N:, LONGITUDE
:1294338E:, MSN-DATE ADD 01:730127:, MSN-NO ADD 01:1199:.
[RECORD ADDED]
```

Figure 11. UDL/ISS Transformation (Update).

(and only) occurrence for the repeating group MSN-DATA, a subscript of 01 is assigned in the ISS ADD statement for all applicable entries.

Record 1 is removed from the file in figure 12. In the case of a DELETE, the "record-ids" must be specified. Therefore, the ISS sequence must output the "record-id" so that ADAPT can use it to actually delete the record.

As shown in figure 13, changes to existing records which do not reference repeating group fields are quite simple. UDL statements map easily onto the ISS change action.

If repeating groups are referenced as in figure 14 or complicated and dependent FIND statements are used as in figure 15, the transformation becomes much more complicated.

```
UDL Statements

     label1 FIND IN TARGET REC-ID EQ 'RECORD-1'
     [NUMBER OF RECORDS FOUND = 1]
     REMOVE SOURCE (label1)

ISS Statements

     REQ NEW
     [ENTER QUERY]
     RECORD-ID = :RECORD-1:.
     ┌NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST┐
     │          TARGET                         1  │
     └ACTION                                      ┘
     OUTPUT S;.
     [RECORD-ID = RECORD-1]
     DELETE
     [INSTRUCTIONS]
     :RECORD-1:.
```

Figure 12. UDL/ISS Transformation (Update).

In figure 14, the records selected must be output so that the actual subscripts of the ISS relational periodics referenced by the UDL repeating groups can be determined. Without the subscripts, a mapping onto ISS is not possible. In the figure, the appropriate subscripts are 03.

In figure 15, the records must be output to determine the "record-ids" to be used in the ISS CHANGE sequence, and to determine the first available subscript for the relational periodic entries to be made from the UDL repeating group. 04 was the first subscript available in this case.

```
UDL Statements

    label1 FIND IN TARGET NAME EQ 'SUBIC BAY'
    [NUMBER OF RECORDS FOUND = 1]
    CHANGE SOURCE (label1) LATITUDE to '144500N', REPORTS
    EQ '465-74' TO '464-74'

ISS Statements

    REQ NEW
    [ENTER QUERY]
    NAME = : SUBIC BAY:.
    [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST
             TARGET                           1
     ACTION                                    ]
    OUT S; RECORD-ID.
    [RECORD-ID = RECORD-2]
    CHANGE
    [INSTRUCTIONS]
    :RECORD-2:, LATITUDE:144500N:, REPORTS CHANGE:465-74:
    TO :464-74:
    [CHANGES ACCEPTED]
```

Figure 13. UDL/ISS Transformation (Update).

DMS-1100/QLP

OVERVIEW – Data Base Management System 1100 (DMS-1100) is Univac's version of CODASYL's DBTG Data Base Management System. The basic premise of the CODASYL Data Base Task Group is that there

```
UDL Statements

     label1 FIND IN TARGET REC-ID EQ 'RECORD-3'

     [NUMBER OF RECORDS FOUND = 1]

     CHANGE SOURCE (label1) OCC (MSN-DATE EQ 760107) MSN-NO

     EQ 2381 TO 2380

ISS Statements

     REQ NEW

     [ENTER QUERY]

     RECORD-ID = :RECORD-3:.

     ⎡NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST⎤
     ⎢           TARGET                      1    ⎥
     ⎣ACTION                                      ⎦

     OUT S; MSN-DATE, MSN-NO, RECORD-ID.

     ⎡OUTPUT LIST                                  ⎤
     ⎣ACTION                                       ⎦

     CHANGE

     [INSTRUCTIONS]

     :RECORD-3:, MSN-NO CHANGE 03:2381: TO 03:2380:.

     [CHANGES ACCEPTED]
```

Figure 14. UDL/ISS Transformation (Update).

```
UDL Statements

     label1 FIND IN TARGET USAGE EQ 'SEAPORT'
     [NUMBER OF RECORDS FOUND = 2]
     label2 FIND SOURCE (label1) MSN-DATA (MSN-DATE EQ 720215
     AND MSN-NO EQ 1104)
     [NUMBER OF RECORDS FOUND = 1]
     ADD SOURCE (label2) MSN-DATA (MSN-DATE EQ 760325,
     MSN-NO EQ 3105)

ISS Statements

     REQ NEW
     [ENTER QUERY]
     USAGE = :SEAPORT:.
     [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
     [          TARGET                          2 ]
     [ACTION                                      ]
     REL NEW
     [ENTER QUERY]
     MSN-DATE = :720215: TO MSN-NO = :1104:
     [NUMBER OF RECORDS WHICH SATISFY YOUR REQUEST]
     [          TARGET                          1 ]
     [ACTION                                      ]
     OUTPUT S; MSN-DATE, MSN-NO, RECORD-ID.
     [OUTPUT LIST     ]
     [     :          ]
     [ACTION          ]
     CHANGE
     [INSTRUCTIONS]
     :RECORD-2:, MSN-DATE ADD 04:760325:, MSN-NO
     ADD 04:3105:.
     [CHANGES ACCEPTED]
```

Figure 15. UDL/ISS Transformation (Update).

must be a separation of data description functions from data access functions. This implies a distinction between the data base manager, whose function is to describe the data records, and the user, whose function is to access the data. This distinction leads to the separate development of a data description language and a data manipulation language.

Currently, DMS-1100 is operational with both the Data Definition Language (DDL) and the Data Manipulation Language (DML). In addition, a new interactive Query Language Processor (QLP) has been recently developed by Univac. The DMS-1100 system resides on the Univac 1100 hardware series where it operates under the EXEC-8 operating system.

DMS-1100 provides great flexibility to the user where simple sequential organization of data to complicated network structures is allowed. These data structures are divided into areas (files), records, and items (fields). In addition, a user may optionally define set relationships over records, where each set is called by name. The record location-mode can be specified by the user where direct, index sequential, calculation by procedure (CALC) and via-set are allowed. A DDL is used by the user to define his data bases.

DMS-1100 provides the user with a DML which is embedded in a Univac 1100 COBOL environment. This language allows the user to interrogate his data base (FIND, FETCH, and GET commands), to update or restructure his data base (STORE, MODIFY, DELETE, INSERT, and REMOVE commands), and to conditionally control the sequencing of his DML procedure (IF command). In addition, other miscellaneous data manipulation, data base control, and run-unit control commands are provided, such as OPEN, CLOSE, MOVE, IMPART, and DEPART. The display capability is provided by the standard COBOL display commands. The online interactive QLP provides essentially the same capabilities as those previously described for the DML. It is this interactive query language that UDL is transformed to.

To illustrate the transformation of DMS-1100 data structures to functionally equivalent UDL data structures, a simple file has been constructed (figure 16). Three records are shown in this file, each containing information concerning a hypothetical employee file. Each record contains fields depicting an employee's name, job position, salary, spouse's name, birthdate, and information concerning the employee's children. The following paragraphs examine this file to show the transformation of DMS-1100 data structures to equivalent UDL data structures, and the mapping of UDL interrogation, display, and update statements to functionally equivalent QLP statements.

DMS-1100 DATA DESIGN –

Data Structures – In figure 16, all major DMS-1100 record data structures are represented in UDL terminology. DMS-1100 areas map onto UDL files. Similarly, the DMS-1100 record maps directly onto a UDL record. DMS-1100 data-items are represented as UDL single-valued fields; i.e., NAME, TITLE, SALARY, and SPOUSE. DMS-1100 array structures are shown with UDL arrays CNAME and PKIND. For the array CNAME, the field CHILDNAM and the array PKIND are defined, and for the array PKIND the field PETS is defined. CHILDNAM and PETS are single-valued fields.

The DMS-1100 UDL data structure mapping is quite straightforward.

Data Types – Of the five data types recognized in UDL, only two are utilized for DMS-1100 files: alphanumeric and numeric. Both of these data types are illustrated in figure 16, where fields NAME, TITLE,

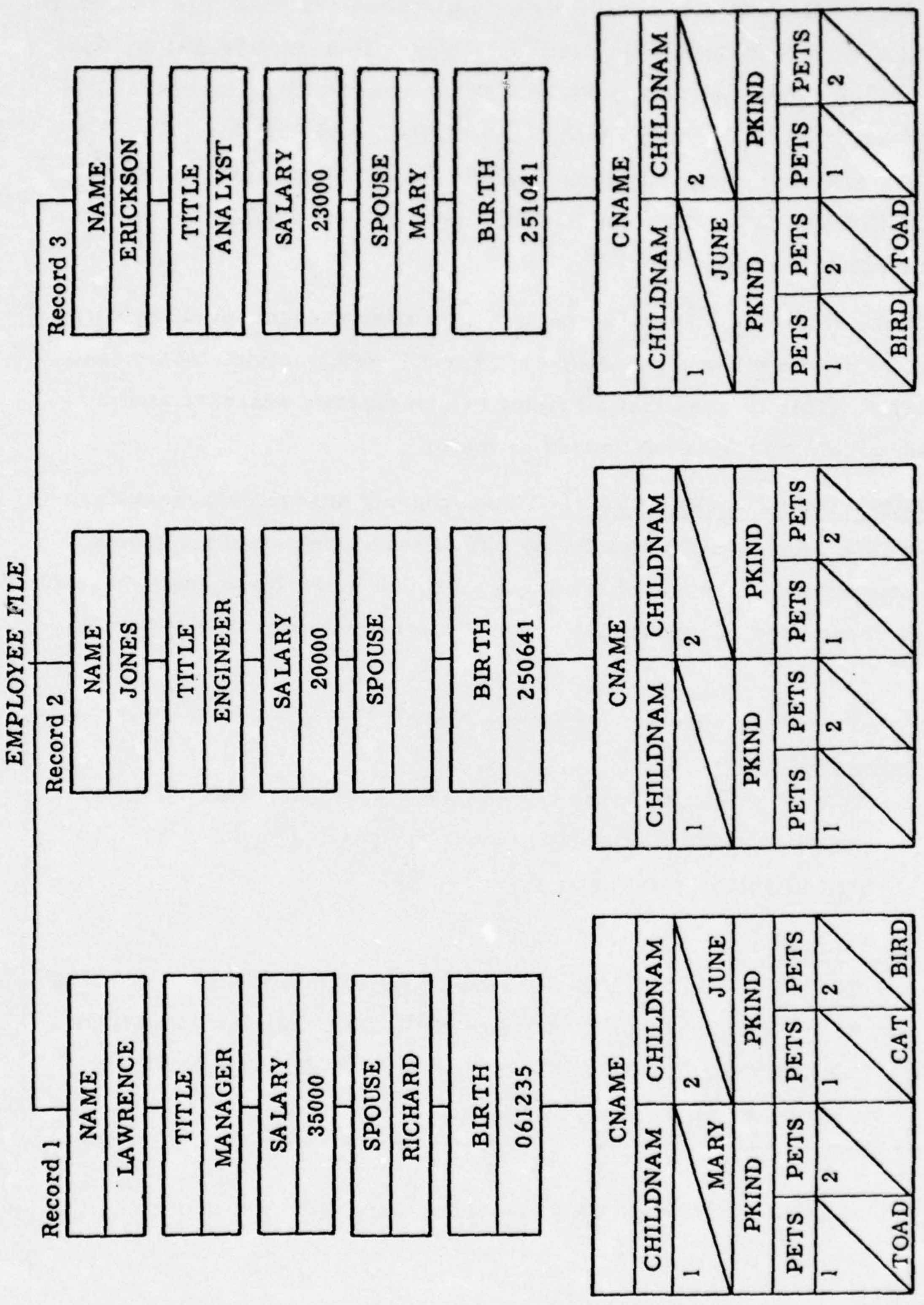


Figure 16. DMS-1100 Sample File.

SPOUSE, CHILDNAM, and PETS have an alphanumeric data type and fields SALARY and BIRTH have a numeric data type. This implies that certain UDL operators can not be applied in FIND statements that reference DMS-1100 files; i. e., CONTAINS, INSIDE, OUTSIDE, and ALONG. The CONTAINS operator is valid only for variable or fixed length text data types, and the INSIDE, OUTSIDE, and ALONG operators are applicable only with geographic data types.

Data Attributes – Of the seven data attributes recognized under UDL, only four will be utilized for DMS-1100 files: keyed, visible, major, and nonmajor. This implies that all fields can be directly searched and displayed. Field NAME is designated as major.

DMS-1100 INTERROGATION – This paragraph describes the transformation of UDL interrogation statements onto a set of functionally equivalent QLP statements. The sample data base utilized in the foregoing paragraph (figure 16) is used in conjunction with the statement sequences shown in figure 17.

UDL Statements

```
label FIND IN EMPLOYEE (SALARY GT 20000) AND (PETS(1, 1)
EQ 'TOAD' XOR NOT CHILDNAM(2) EQ 'JUNE')
[NUMBER OF RECORDS FOUND = 2]
```

QLP Statements

```
COUNT SALARY PETS CHILDNAM WHERE (SALARY GT '20000'
AND PETS(1, 1) EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE') OR
(SALARY EQ '20000' AND PETS(1, 1) NOT EQ 'TOAD' AND
CHILDNAM(2) NOT EQ 'JUNE')
```

Figure 17. UDL/QLP Transformation (Interrogation).

Consider the UDL statement sequence shown in figure 17. The UDL FIND statement isolates those employees whose salary exceeds \$20,000, where this employee either does not have a second child named June or does have a first child that owns a pet toad, but not both conditions. Employees Lawrence and Erickson satisfy this FIND statement (records 1 and 3).

The functionally equivalent QLP statements are shown in figure 17. The QLP COUNT statement is used to get the number of records satisfying the selection criteria specified in the where-clause. Since QLP does not have an exclusive-or operator (XOR), a logically equivalent expression using AND, OR, and NOT operators is used. The QLP COUNT statement provides counts of records for each field named in the specified field list.

The UDL/QLP interrogation statement transformation is quite straightforward.

<u>QLP DISPLAY</u> – This paragraph describes the transformations required for requesting partial and whole records from DMS-1100 for subsequent display through UDL. Figure 16 is used as the sample data base for illustrating these transformations.

Consider the UDL statement sequence in figure 18. The FIND statement isolates two employee records, record 1 and record 3. The DISPLAY statement outputs the employee's name and the employee's spouse's name for each of these records in the UDL default format. Although in UDL a user can reference a previous interrogation statement remotely via a statement label, the selection criteria must be respecified in the where-clause for the QLP LIST statement. For the default case where the display-list is not specified in the UDL DISPLAY statement, UDL will output the entire record. In this case, all fields must be specified in the QLP LIST statement.

<u>QLP UPDATE</u> – This paragraph discusses the transformation of UDL update statements to equivalent QLP statement sequences. As in the

UDL Statements

```
label FIND IN EMPLOYEE (SALARY GT 20000) AND (PETS(1, 1)
EQ 'TOAD' XOR NOT CHILDNAM(2) EQ 'JUNE')
[NUMBER OF RECORDS FOUND =2]
DISPLAY SOURCE (label) NAME, SPOUSE

    [... record header ...
     NAME = LAWRENCE
     SPOUSE = RICHARD
     ... record header ...
     NAME = ERICKSON
     SPOUSE = MARY]
```

QLP Statements

```
COUNT SALARY PETS CHILDNAM WHERE (SALARY GT '20000'
AND PETS(1, 1) EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE')
OR (SALARY EQ '20000' AND PETS(1, 1) NOT EQ 'TOAD'
AND CHILDNAM(2) NOT EQ 'JUNE')

LIST NAME SPOUSE WHERE (SALARY GT '20000' AND PETS(1,1)
EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE') OR (SALARY GT
'20000' AND PETS(1, 1) NOT EQ 'TOAD' AND CHILDNAM(2)
NOT EQ 'JUNE')
```

Figure 18. UDL/QLP Transformation (Display).

preceding, the sample data base presented in figure 16 is used in conjunction with the statement sequences shown in figures 19 through 21.

Of the five UDL update statements available, only three can be used on DMS-1100 files. Since DMS-1100 does not provide true multiply occurring data structures below the record level, the UDL ADD or DELETE statements are not legal. Although DMS-1100 provides array structures which may contain multiple values, array structure sizes are fixed at definition time, hence they can not be expanded or reduced.

UDL Statements

```
label FIND IN EMPLOYEE (SALARY GT 20000) AND (PETS(1, 1)
EQ 'TOAD' XOR NOT CHILDNAM(2) EQ 'JUNE')
[NUMBER OF RECORDS FOUND = 2]
REMOVE SOURCE (label)
```

QLP Statements

```
COUNT SALARY PETS CHILDNAM WHERE (SALARY GT '20000'
AND PETS(1, 1) EQ 'TOAD' AND CHILDNAM(2) EQ 'JUNE')
OR (SALARY EQ '20000' AND PETS(1, 1) NOT EQ 'TOAD' AND
CHILDNAM(2) NOT EQ 'JUNE')

DELETE EMP WHERE (SALARY GT '20000' AND PETS(1, 1) EQ
'TOAD' AND CHILDNAM(2) EQ 'JUNE') OR (SALARY GT
'20000' AND PETS(1, 1) NOT EQ 'TOAD' AND CHILDNAM(2)
NOT EQ 'JUNE')
```

Figure 19. UDL/QLP Transformation (Update).

UDL Statements

```
CREATE IN EMPLOYEE NAME EQ 'JOHNSON', TITLE EQ
'PROGRAMMER', SPOUSE EQ 'MARY', BIRTH EQ 070645,
CHILDNAM(1) EQ 'WILLIAM', PETS(1, 1) EQ 'DOG'
```

QLP Statements

```
CREATE EMP WITH NAME EQ 'JOHNSON', TITLE EQ
'PROGRAMMER', SPOUSE EQ 'MARY', BIRTH EQ '070645',
CHILDNAM(1) EQ 'WILLIAM', PETS(1, 1) EQ 'DOG'
```

Figure 20. UDL/QLP Transformation (Update).

UDL Statements

```
label FIND IN EMPLOYEE NAME EQ 'ERICKSON'
[NUMBER OF RECORDS FOUND = 1]
CHANGE SOURCE (label) SALARY TO 26000, CHILDNAM(2)
TO 'MICHAEL', PETS(2, 1) TO 'HORSE'
```

QLP Statements

```
COUNT NAME WHERE NAME EQ 'ERICKSON'
CHANGE SALARY EQ '26000' CHILDNAM(2) EQ 'MICHAEL'
PETS(2, 1) EQ 'HORSE' WHERE NAME EQ 'ERICKSON'
```

Figure 21. UDL/QLP Transformation (Update).

In figure 19, all employees who satisfy the conditions specified in the FIND statement are removed from the employee file. The corresponding QLP statement sequence must repeat the selection criteria in the DELETE statement since QLP does not provide a facility for remotely specifying interrogation sequences. In figure 20, a new employee named JOHNSON is added to the employee file. The transformation to an equivalent QLP statement sequence is very straightforward, where the same comments concerning figure 19 also apply.

In figure 21, employee Erickson's salary is changed from $23,000 to $26,000 and a new child and its associated pet is added. Although a new child and pet are added, the array position (index 2) was already available, hence the modification can be accomplished with the UDL CHANGE statement. In figure 16, the arrays CNAME and PKIND have two index positions each; therefore, a user could not add a third child or pet.

SOLIS

OVERVIEW – Even though the SIGINT On-Line Information System (SOLIS) is a document retrieval system, the data structures of SOLIS follow the standard structures defined in UDL. Therefore, the interrogation of the SOLIS data base is easily accomplished in UDL.

A sample document file has been described for SOLIS in UDL format and terminology and is shown in figure 22. There are three records (documents) in the file, each with a title, message number, date, and text. An additional field is shown called TEXTTERM which contains the actual key terms in the text of the document which can be used in forming queries.

An important point to notice in this sample data base is the mapping of the field names used in UDL to the actual retrieval strategies in SOLIS. For example, the field called TITLE in UDL actually corresponds to

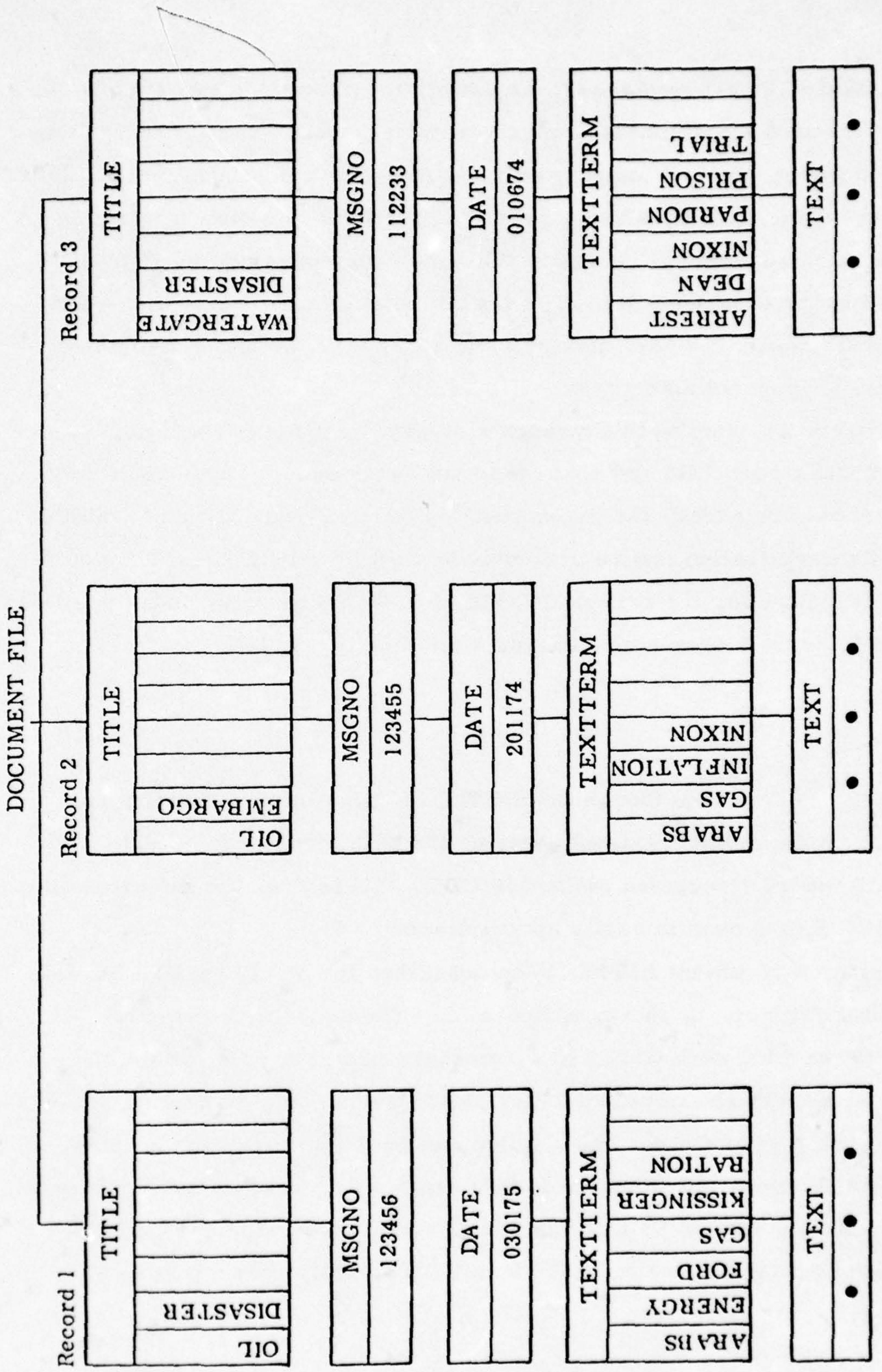


Figure 22. SOLIS Sample File.

TILDEX in SOLIS. This mapping provides the uninitiated user with a more meaningful "retrieval strategy" than provided in SOLIS.

SOLIS provides online editing, storage, and retrieval of NSA end-product documents. The most current six months of documents are online with the newest document being at least 15 minutes old. Documents are retrieved using terms, recognizable by SOLIS as index terms, which are kept in one or more system dictionaries. SOLIS has been operational since November, 1972.

SOLIS resides on the Burroughs B-6700 computer system. The remote terminals are Burroughs B-9353 CRTs and teletypewriters for low-speed, hardcopy output. High-speed hardcopy output takes place near the B-6700 on high-speed printers.

Messages enter the SOLIS system in a raw, unretrievable form. Each message is filed, edited, and made retrievable. Old messages are automatically phased out of SOLIS as new messages come in. All editing and retrieval functions are handled by a software package called the SOLIS/MESSENGER. SOLIS physical data structures are organized by volume, file, record, and word.

The logical data entities are terms from the document and the documents themselves.

Document retrieval and display are accomplished by the user entering the parts of his request on a formatted CRT screen. Basically, key words and/or other document descriptors (such as date of entry into SOLIS) are entered on the CRT and the Send key is depressed. The response from SOLIS is the number of documents satisfying the request. To then view the text of the documents retrieved, the user need only press the Send key again and the zero page of the message will appear.

A guide to the sample field name-retrieval strategy mapping is shown in figure 23.

UDL Field Names	SOLIS Retrieval Strategy	Meaning
TITLE	TILDEX	(Long title)
MSGNO	S	(Serial number)
DATE	D	(Date)
TEXTTERM	WINDEX	(Terms from the message text)

Figure 23. SOLIS Retrieval Strategy Mappings.

Detailed descriptions of the UDL data structures found in SOLIS and queries to this data base follow.

SOLIS DATA DESIGN –

Data Structures – There are two UDL field structures found in SOLIS, both illustrated in the sample data base (figure 22). One is the single-valued field and is found in the fields named MSGNO, DATE, and TEXT. The other is the multivalued field represented by the fields TITLE and TEXTTERM.

The record data structure in UDL corresponds to the document structure in SOLIS. In other words, when the user, using UDL, selects a record from the SOLIS data base, he is actually selecting a SOLIS document. UDL will also map individual files onto the various volumes that are distinguished in SOLIS. UDL files correspond to SOLIS file/volume pairs.

Data Types – All the fields in the SOLIS sample file, except DATE, are considered to be character type; specifically, alphanumeric. DATE is a numeric field.

Data Attributes – In SOLIS, only the actual document text and a few preamble fields can be displayed and, therefore, the only "visible" fields are TEXT and MSGNO. But it is possible to query the system based on selected parts of each document, such as the title (TITLE), the message number (MSGNO), the date (DATE), or terms in the text of the document (TEXTTERM). Therefore, these fields are "keyed." The TEXT field is "nonkeyed" because not all terms in the text of the document can be used in selection criteria; i. e., only those in the field TEXTTERM can be used in selection criteria. Figure 24 shows the fields from the sample file and their attributes.

SOLIS INTERROGATION – Figure 25 illustrates how a UDL FIND statement would access the SOLIS document file (figure 22) and how this would be accomplished via translation to the SOLIS query language format. The problem is to determine how many records (documents) satisfy the condition that the term GAS is in the text of the document but the term DISASTER does not appear in the title of the document. One record is selected (record 2) by this FIND statement, and the UDL response indicates this (note the response in brackets). In all cases of mapping from UDL to SOLIS, the "free screen" will be established prior to transmitting the SOLIS query. The response to the SOLIS query is the number of documents satisfying the query (i. e., "1 SIGINT ITEMS SATISFY THIS QUERY"), and this obviously can be used directly to generate the appropriate UDL response.

Field	Attributes
TITLE	Keyed, invisible
MSGNO	Keyed, visible
DATE	Keyed, invisible
TEXTTERM	Keyed, invisible
TEXT	Nonkeyed, visible

Figure 24. SOLIS Field Attributes.

```
UDL Statements

    FIND IN DOCUMENT TEXTTERM EQ 'GAS' AND NOT TITLE
    EQ 'DISASTER'
    [NUMBER OF RECORDS FOUND = 1]

SOLIS Statements

    (free screen established)
    WINDEX GAS WITHOUT TILDEX DISASTER
```

Figure 25. UDL/SOLIS Transformation (Interrogation).

SOLIS DISPLAY – Figure 26 includes the same query as in the interrogation example. The task is to display all the records selected (namely one), and of these records to display the field named TEXT. The output from this UDL DISPLAY statement is shown in the large brackets, and is

```
UDL Statements

    label FIND IN DOCUMENT TEXTTERM EQ 'GAS' AND NOT
    TITLE EQ 'DISASTER'
    [NUMBER OF RECORDS FOUND = 1]
    DISPLAY SOURCE (label) TEXT
    [... record header ...          ]
    [TEXT = ... text of document    ]

SOLIS Statements

    (free screen established)
    WINDEX GAS WITHOUT TILDEX DISASTER
    (Send key used to retrieve each page of document found)
```

Figure 26. UDL/SOLIS Transformation (Display).

obtained from SOLIS via simulating the transmission of the Send key. The Send key will retrieve the entirety of each document page by page.

By specifying "PREAMBLE AO," UDL can also receive from SOLIS selected document information such as the DTG, long title, and the serial number. These will be interpreted as visible fields in UDL.

TIPS/TILE

OVERVIEW – The Technical Information Processing System (TIPS) was conceived in 1960. By 1965, TIPS was operational on two Univac 490 systems, handling user requests on a 24-hour, seven-day week basis. This was a somewhat restricted system, however, handling only one request at a time and having no standard query language available for the various user data files. Presently, TIPS runs on three Univac 494 computers, where one is completely dedicated for TIPS processing. A standard query language is now available, called the TIPS Interrogation Language (TILE). Connected to the Univac 494 are 24 Honeywell 516 minicomputers. Each 516 can handle a separate query simultaneously, thus providing TIPS with simultaneity of operation. TIPS operates under the RYE operating system.

The files accessed with TILE are either single-format or multiformat files. The records within both types of files are fixed size. Within the single-format file, all records have the same format, contain the same fields, and are the same size. In a multiformat file, up to 60 formats may be used. Files, formats, and fields are called by name.

The TILE language is composed of statements. TILE statements normally begin with verbs followed by nouns specifying files, fields, and data, related by connectors and relations. The verbs available under TILE include: EXT (extract), PRINT, SORT, CREATE, POST, USE, BUILD, PUBLISH, CONST, INVOK, and UPD. PRINT has its obvious meaning. SORT allows the user to sort the output of an extract in some order other than that which is maintained by the system. CREATE and POST allow the user to create a temporary subfile. USE allows the user to leave the TILE language for special-purpose functions and return to the TILE language. BUILD allows the user to structure canned print format specifications which can be activated by the PUBLISH verb for subsequent printing. CONST gives the user the ability to construct TILE statement strings which can be activated by the INVOK verb. UPD is the update function for files.

A sample data base has been established for TILE in UDL format and terminology (see figure 27). There are three records, each describing a particular employee. Each record contains information such as employee name, title, salary, spouse, birthdate, children's names, and birthplace.

TILE DATA DESIGN –

Data Structures – There are two field structures in TILE: single-valued fields and multivalued fields which correspond to the TILE family field. These are illustrated in figure 27. CHILDNAM is a multivalued field; the others are single-valued.

The record in UDL corresponds to the record structure in TILE. UDL files correspond to TILE files, and likewise for data bases.

Data Types – Field types in TILE are numeric, alphanumeric, or fixed-length text. The sample data base SALARY is numeric, BIRTHPLA is fixed-length text, and the other fields are alphanumeric.

Data Attributes – All fields in the sample data base are keyed and visible. For actual TILE files, all fields will be keyed and visible. The field NAME is designated major, the others are nonmajor.

TILE INTERROGATION/DISPLAY – Figure 28 illustrates a relatively straightforward example of some mappings of operators in UDL to operators in TILE. The objective of the FIND statement is to select all records in

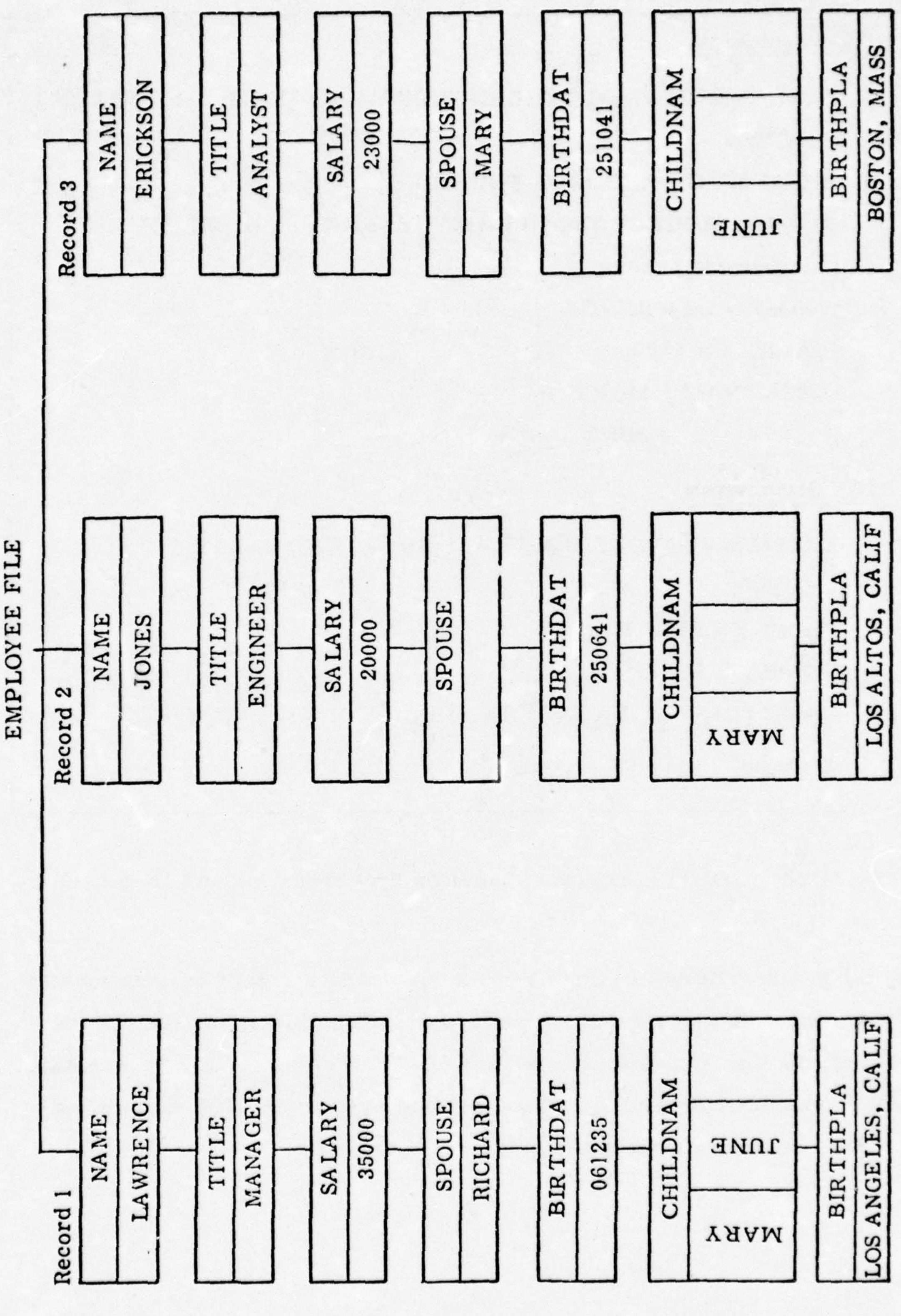


Figure 27. TILE Sample File.

UDL Statements

```
label FIND IN EMPLOYEE CHILDNAM EQ 'JUNE' AND SALARY
GT 23000
[NUMBER OF RECORDS FOUND = 1]
DISPLAY SOURCE (label) NAME, SALARY, CHILDNAM
[... record header ...    ]
[NAME = LAWRENCE          ]
[SALARY = 35000           ]
[CHILDNAM = MARY          ]
[         = JUNE          ]
```

TILE Statements

```
EXTI/EMPLOYEE; CHILDNAM [JUNE] AND SALARY
> (23000).
PRINT EMPLOYEE:
FORMAT: PART
NAME (1 to X1), SALARY (X1 + 2 to X2), CHILDNAM (X2 + 2
TO X3).
```

Figure 28. UDL/TILE Transformation (Interrogation and Display).

which there is a child named June and the employee's salary is greater than $23,000. One record is selected (record 1). The display of fields from all records is also shown in this example. The fields displayed are NAME, SALARY, and CHILDNAM, and they are displayed in the UDL default format. The output is shown in the large brackets.

TILE UPDATE – The update example shown in figure 29 illustrates a change of the SALARY field from 35000 to 38000. Notice that the selection criteria had to be repeated in the TILE update command since there is no command dependency facility in TILE.

Figure 30 illustrates the removal of employees from the employee file. In this example, all employees named JONES (record 2) are to be removed. For the equivalent statement sequence in TILE, the DEL ALL subfunction of the UPD statement is utilized. Note that the file-name EMPLOYEE must be specified as well as the selection criteria. Figure 31 shows how a new employee, Gonzales, is added to the employee file. The TILE ADD sub-function is used as a transformation where each field initialization is specified.

UDL Statements

```
label FIND IN EMPLOYEE CHILDNAM EQ 'JUNE' AND
SALARY GT 23000
[NUMBER OF RECORDS FOUND = 1]
CHANGE SOURCE (label) SALARY TO 38000
```

TILE Statements

```
EXTI/EMPLOYEE; CHILDNAM [JUNE] AND SALARY > (23000).
UPD/EMPLOYEE; SUB ALL CHILDNAM [JUNE] AND SALARY
> (23000) TO SALARY (38000).
```

Figure 29. UDL/TILE Transformation (Update).

```
UDL Statements

    label FIND IN EMPLOYEE NAME EQ 'JONES'
    [NUMBER OF RECORDS FOUND = 1]
    REMOVE SOURCE (label)

TILE Statements

    EXTI/EMPLOYEE; NAME [JONES].
    UPD/EMPLOYEE; DEL ALL NAME [JONES].
```

Figure 30. UDL/TILE Transformation (Update).

```
UDL Statements

    CREATE IN EMPLOYEE NAME EQ 'GONZALES', TITLE
    EQ 'ENGINEER', SALARY EQ 25000, CHILDNAM EQ 'JOSE'

TILE Statements

    UPD/EMPLOYEE; ADD NAME [GONZALES] AND TITLE
    [ENGINEER] AND SALARY [25000] AND CHILDNAM [JOSE].
```

Figure 31. UDL/TILE Transformation (Update).

ADAPT I ENVIRONMENT

This section presents a general picture of the ADAPT I system as it operates in a UNIX environment, and should be read before one attempts to digest the other sections describing the various ADAPT I processes and global data structures. It is also recommended that "ADAPT I Uniform Data Language (UDL): A Preliminary Specification," 23 July 1976 be consulted before reading this section. There are many references in this section, as well as in other sections, to the many UDL/DDL statements and commands provided under ADAPT I and, therefore, a reasonable understanding of these language constructs will greatly enhance the reading of this document.

The material presented in this section has been divided into three parts: 1) a somewhat detailed description of the ADAPT I file environment is provided, 2) a brief description of the ADAPT I System Generation Program is given, and 3) a general narrative-type discussion illustrating the overall communication paths which are exercised by the various ADAPT I processes is provided. After reading this section one should have a general understanding of the overall system architecture comprising ADAPT I.

ADAPT I File Environment

One design goal that has been constantly pursued was to utilize the hierarchical file structures supported under the UNIX operating system. Although many of the ADAPT I global data files are "internally" structured, they have been embedded in a somewhat elaborate UNIX hierarchical file structure.

The ADAPT I file environment described in the following is the entire data set over which all ADAPT I processes operate. This data set is composed of UNIX directories and ordinary files. Some of the directories have a fixed number of entries, while others are added to or deleted from dynamically by ADAPT I processes. Some of the ordinary files, most of which adhere to some internal structuring imposed by ADAPT I, always exist in this environment although they may change in size. Other ordinary files may or may not exist depending on what tasks ADAPT I is performing.

Figure 32 illustrates the entire ADAPT I file environment currently envisioned for ADAPT I. The "root" of the ADAPT I file environment is currently attached to the "usr" directory of UNIX. That is, the root path name beginning all ADAPT I file references is "/usr/adapt." Currently, the "/usr/adapt" directory contains nine permanent entries: file pointers to four structured global files, and pointers to five permanent directories. In addition to the nine permanent entries, the root directory "/usr/adapt" will also contain entries representing three "lock-files." Currently ADAPT I has three global data files which are "read-for-update" sensitive. Three lock-files (lock1, lock2, and lock3; path names "/usr/adapt/lock<u>i</u>" where i = 1, 2, or 3) are associated with these global files, their creation and deletion being controlled by two global functions GFLOCK and GFUNLOCK (refer to the last section of this specification). These lock-files are directory entries only, hence do not actually occupy space as files.

The four permanent global files pointed to by "/usr/adapt" are the user description file, GUSER (path name "/user/adapt/guser"); the file description file, GFILD (path name "/usr/adapt/gfild"); the file name file, GFILN (path name "/usr/adapt/gfiln"); and the logical transaction description file, GTDES (path name "/usr/adapt/gtdes"). GUSER contains an entry for each user registered under ADAPT I (refer to the following discussion of the ADAPT I Sysgen Program) and is only increased or decreased in size if a

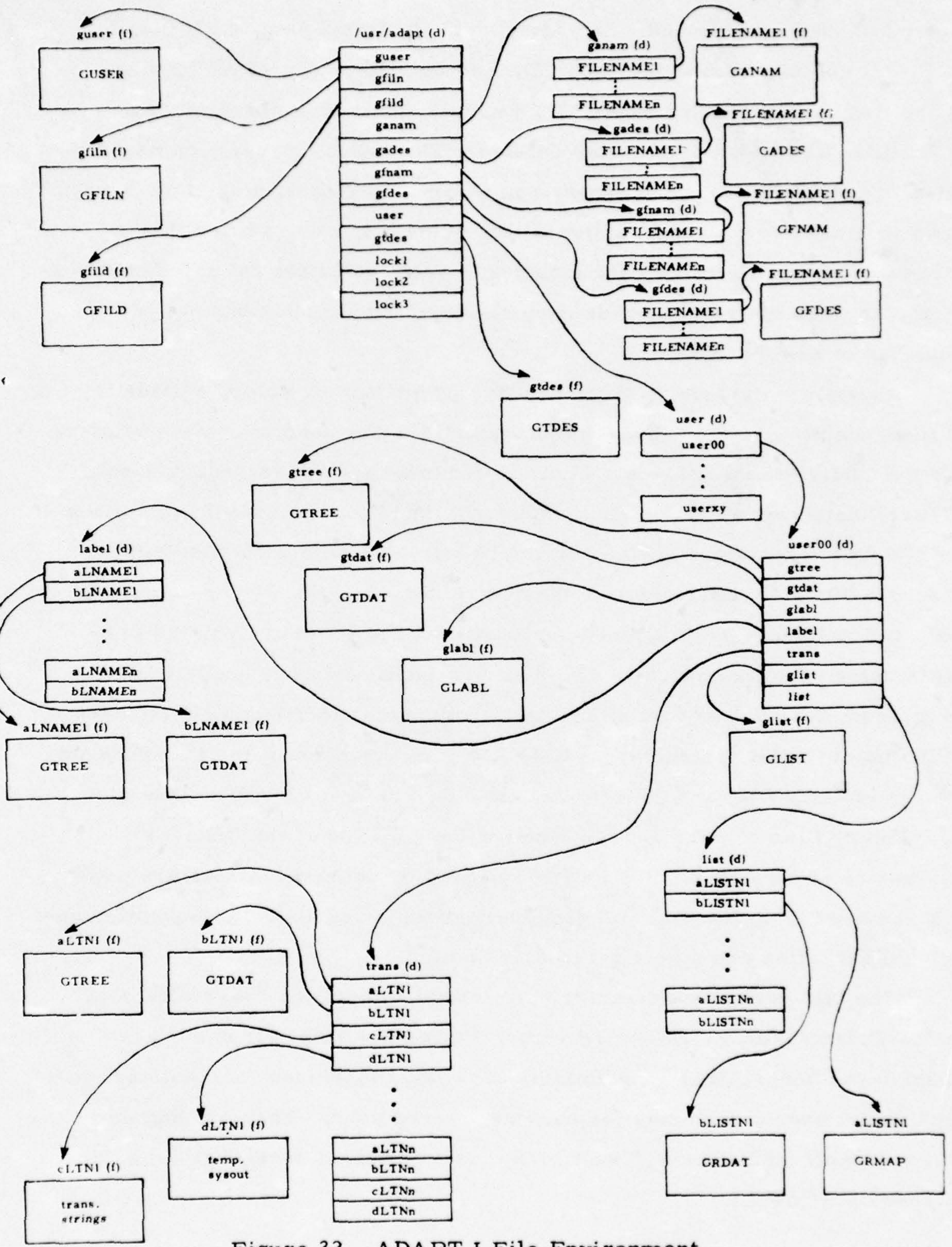


Figure 32. ADAPT I File Environment.

user is added or removed. Similarly, the file global data, GFILD and GFILN, contain entries for each UDL file defined within ADAPT I (i. e., files that can be queried by ADAPT I users), their size changing as new UDL files are added or old files deleted. The logical transaction description file, GTDES, is more dynamic in nature, only containing entries when transactions are currently active within ADAPT I; i. e., when either a transaction is still waiting for a response from some distant batch host system, or when the response has been received but the user has not yet requested to see the output.

As stated previously, there are five permanent directory entries in "/usr/adapt." Four of these directory entries are used to contain pointers to the UDL file dictionaries. Their path names are "/user/adapt/ganam," "/usr/adapt/gades," "/usr/adapt/gfnam," and "/usr/adapt/gfdes." Each of the four directories contains a single entry for each UDL file defined under ADAPT I. Each entry in these directories is the UDL-name of the file and points to a structured global file. The structure pointed to by path name "/usr/adapt/gades/filex" is the global data file GADES, the aggregate description file. Similarly, path names to the other dictionary files point to global structures: GANAM, the aggregate name file; GFNAM, the field name file; and GFDES, the field description file. The four global dictionary files remain fixed in size for the duration of the UDL file's existence under ADAPT I. If a file is added or deleted, entries are made or removed from the four file dictionary directories and, subsequently, new global data files are either created or deleted.

The last permanent directory in "/usr/adapt" is the "user directory," "/usr/adapt/user." This directory contains one entry for each user registered under ADAPT I. Entries in "/usr/adapt/user" are pointers to individual user directories for each registered user. They are named "/usr/adapt/user/userxy," where "xy" is an internal user-id (UID) established by ADAPT I.

Each user directory can contain up to seven entries: four global data file pointers and three permanent directory pointers. Two of the global data entries point to the parse tree representation of a UDL (or DDL) statement or command that the user has entered through his terminal. Parse tree global data files, GTREE and GTDAT, are created/deleted each time a user enters a statement or command. Therefore, files "/usr/adapt/user/userxy/gtree" and "/usr/adapt/user/userxy/gtdat" do not exist unless the user (identified by an UID equal "xy") is currently logged on to ADAPT I. The UDL statement label file, GLABL, is pointed to the path name "/usr/adapt/user/userxy/glabl." This global data file contains the statement labels for that user during a current logon session. Therefore, GLABL is similar to the parse tree files, GTREE and GTDAT, since it only exists while a given user is logged on to ADAPT I. The user list description file, GLIST, is pointed to by the user directory and has the path name "/usr/adapt/user/userxy/glist." Unlike the label file, this global data file may exist whether the user is logged on or not. It contains descriptions of user-defined lists (explicit lists) and possible implicit lists created internally by ADAPT I for active transactions. Its size may increase or decrease, depending on how many lists the user has defined.

There are three permanent directory entries found in the individual user directories: one pointing to UDL list record data, "usr/adapt/user/userxy/list"; one pointing to parse tree data represented by a statement label, "/usr/adapt/user/userxy/label"; and one pointing to information representing a logical transaction; "/usr/adapt/user/userxy/trans." The user list record-data directory contains two entries for each list described in GLIST. The first entry points to a global data file structured as a set of UDL record maps, GRMAP; the second entry points to a global data file structured as a set of UDL record data, GRDAT. This implies that all file records contained locally in ADAPT I are associated with a list-name. The naming convention utilized to differentiate the two entries in "/usr/adapt/user/userxy/list" is as follows. The alphabetic character "a" is concatenated to the list-name as a prefix to designate GRMAP (i. e., "/usr/adapt/user/userxy/list/alistname") and the alphabetic character "b" is concatenated as a prefix to designate GRDAT. Implicit list-names are generated internally by ADAPT I when the user requests the display of record data from some remote file residing on a distant host system. This request can occur in two ways:

a. The remote file may reside on a "batch" host system and therefore a logical transaction must be initiated to that host system. This transaction is assigned a logical transaction number (LTN, $1 \leq \text{LTN} \leq 255$) unique to that user. The alphabetic character "a" concatenated to the LTN is used to form the implicit list name as it is actually inserted into GLIST and "aaLTN" and "baLTN" are the ending file designators in the path names for the associated record information, GRMAP and GRDAT.

b. The remote file may reside on an "interactive" host system. Since an ADAPT I user can only converse with one "interactive" system at any one time, the logical transaction number, 000, has been reserved for interactive queries. Therefore, "a000" is used as the implicit list-name and "aa000" and "ba000" are used to designate the record global data files.

The user label directory, "/usr/adapt/user/userxy/label," is required to point to parse tree information representing the UDL statement to which the label was affixed. These global data are GTREE and GTDAT. For each label entry in GLABL there exist two entries in the user list directory, one pointing to GTREE and the other pointing to GTDAT. The naming convention for this directory is to concatenate the alphabetic "a" to the label-name to designate GTREE (i. e., "/usr/adapt/user/userxy/list/alabelname") and to concatenate "b" to the label-name to designate GTDAT. Incidently, the original parse tree global data files that reside at "/usr/adapt/user/userxy/gtree" and ". . . /gtdat" are linked to form these entries in the user label directory. As stated earlier, when the user logs-off from ADAPT I, all GLABL entries are removed and consequently so are entries in the user label directory.

The last user directory is the user transaction directory, "/usr/adapt/user/userxy/trans." This directory is used primarily for controlling the batch transactions a user initiates. The transaction directory contains four entries for each logical transaction currently active for a user (it is empty if the user has no outstanding transactions); optional parse tree global data, GTREE and GTDAT, representing a DISPLAY statement; a "transformation string" file which also doubles as a "response string" repository; and a "temporary output" file to contain the final user output prior to his perusal. The naming conventions utilized for these four directory entries are the concatenation of the alphabetic characters "a," "b," "c," and "d" to the logical transaction number: "aLTN" designates GTREE, "bLTN" designates GTDAT, "cLTN" designates the "transformation/response string" file, and "dLTN" designates the "temporary output" file. For interactive transactions (LTN of "000"), only the "transformation/response string" file is required; i.e., "/usr/adapt/user/userxy/trans/c000." The other three entries are not required because the display responses are occurring while the user is "connected" to that host system.

The ADAPT I System Generation (Sysgen) Program

Due to the somewhat complicated UNIX file environment utilized by ADAPT I, it will be necessary to provide an "interactive" Sysgen Program which will generate this hierarchical file structure. This program must be initiated by the UNIX superuser since directories are being created as well as deleted. The ADAPT I Sysgen Program will operate in two modes:

a. "Initial" mode – creates the baseline UNIX file structure.

b. "Update" mode – allows the registration and/or deletion of ADAPT I users.

The "initial" mode will establish the following permanent directories and files for the ADAPT I environment:

a. /usr/adapt (directory).
b. /usr/adapt/guser (file).
c. /usr/adapt/gfiln (file).
d. /usr/adapt/gfild (file).
e. /usr/adapt/ganam (directory).
f. /usr/adapt/gades (directory).
g. /usr/adapt/gfnam (directory).
h. /usr/adapt/gfdes (directory).
i. /usr/adapt/user (directory).
j. /usr/adapt/gtdes (file).

Upon the generation of baseline UNIX files and directories, the ADAPT I Sysgen Program requests the identification of the ADAPT I superuser. Upon entry of this identifier, the Sysgen Program creates the following directories and files for the ADAPT I superuser:

a. /usr/adapt/user/user00 (directory).
b. /usr/adapt/user/user00/trans (directory).
c. /usr/adapt/user/user00/glist (file).
d. /usr/adapt/user/user00/list (directory).
e. /usr/adapt/user/user00/label (directory).

The Sysgen Program then transfers to the second mode (which can be entered separately). The second mode "converses" with the UNIX superuser and prompts him to enter the next ADAPT I user. At this point, new ADAPT I users (who must be unique within ADAPT I) can be registered or deleted, a new ADAPT I superuser can be assigned, or the program can be terminated. For each new user registered, the directories and files which were created for the superuser are created for that user's ID.

As stated previously, the second mode of the ADAPT I Sysgen Program can be entered separately. In this case, it is assumed that a baseline ADAPT I environment does exist. The second mode is used exclusively for the registration or deletion of ADAPT I users.

As implied by the foregoing discussion, the ADAPT I system requires a superuser. This user has special privileges not available to the ADAPT I users. For example, only the ADAPT I superuser can define a file SCHEMA or delete a file SCHEMA. The ADAPT I superuser is the only user who can request a report on all transactions currently active within ADAPT I. Nonsuperusers can only request reports on their own active transactions.

ADAPT I Processes

The term "process" is used herein in the literal sense as in UNIX; i.e., an executable file (program) which can be initiated either through terminal commands entered by a user who has "execute access rights," or by another executing process via the "execute" system call.

Figure 33 illustrates 11 processes currently envisioned for ADAPT I, showing their communication paths with each other, with the user at the terminal, and with the COINS II Network. Also provided in figure 33 is some general association of the UDL/DDL statement/command set with the process(es) which perform the appropriate operation.

Probably the best way to illustrate the interaction that occurs between the different ADAPT I processes is to present a set of simple "user scenarios" involving user statement/command entries and corresponding network responses. The scenarios presented in the following are not necessarily complete with respect to actual user statement entries; however, they do provide a general framework in which to view the operation of ADAPT I.

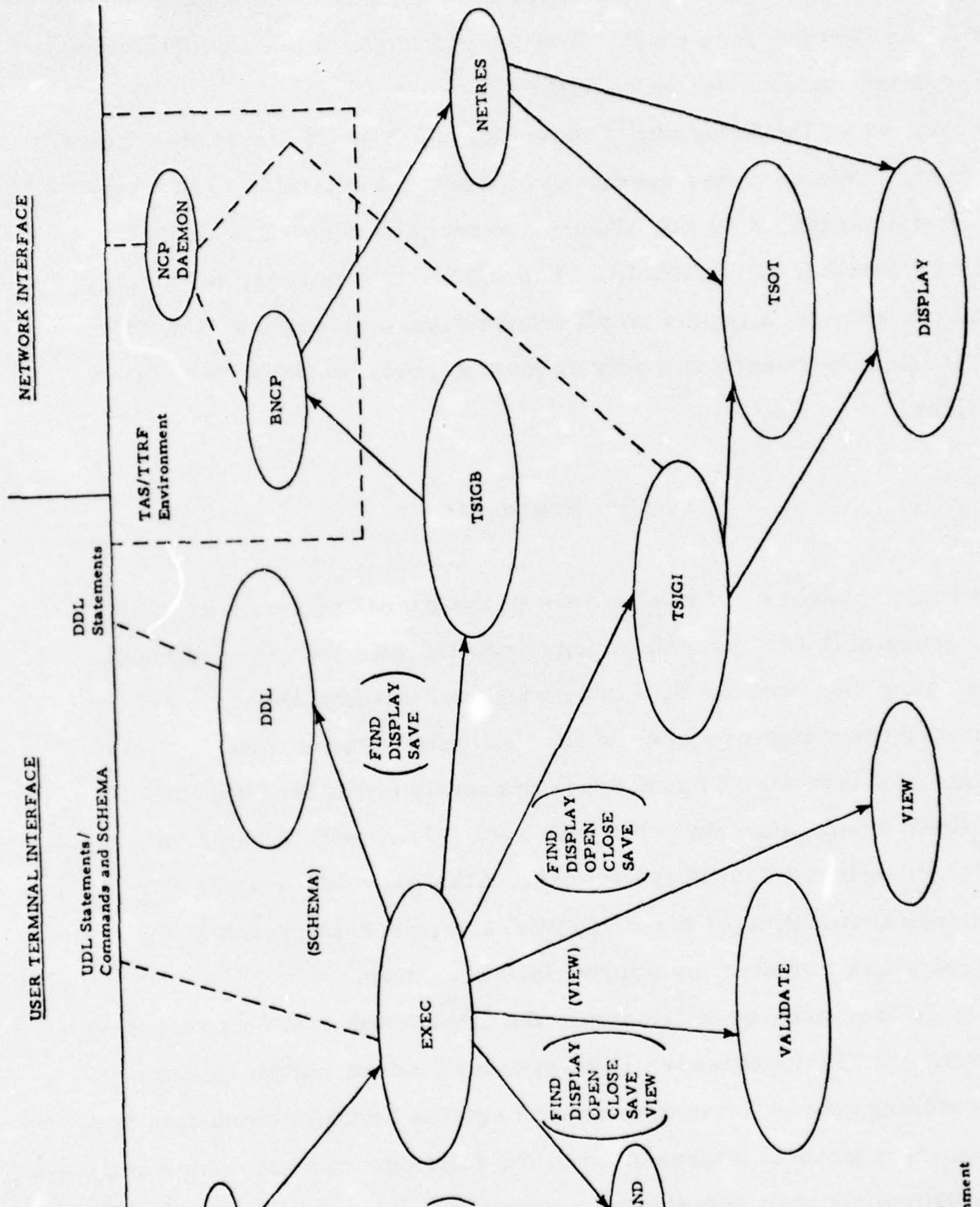


Figure 33. ADAPT I Process Communication.

The discussion which follows will be concerned with four simple scenarios:

a. The ADAPT I superuser logs-on to ADAPT I and describes a new file via the DDL sublanguage.

b. An ADAPT I user (not necessarily the superuser) queries an "interactive" file and displays a set of fields from the selected record set.

c. An ADAPT I user queries a "batch" file and saves the selected records on a list.

d. An ADAPT I user views a specified file schema and then logs-off from ADAPT I.

SCENARIO 1 – For users (any user) to logon successfully to ADAPT I, they must enter the ADAPT command through a terminal. The UNIX shell, which recognizes this command, activates the process called ADAPT. ADAPT is a small process which performs user validation on the ADAPT command. If the user is an authorized ADAPT user, hence has been registered by the ADAPT I Sysgen Program, ADAPT activates the ADAPT I executive process, EXEC. From this point on all user/terminal interfaces are either directed entirely from EXEC, or at least initiated from EXEC. The user can now proceed with other UDL commands or statements and, if he or she is the superuser (superuser in the ADAPT sense), can also utilize the Data Definition Language. The EXEC process performs the "front-end" processing of all UDL statements and commands and the SCHEMA statement. Depending on the statement or command, EXEC may process it itself or call one or more other processes for processing.

Assume that the ADAPT I superuser has logged on to ADAPT I and enters the SCHEMA statement. Since this is the superuser, EXEC will call the DDL process for processing of the schema definition. The DDL process is responsible for generating UDL file dictionaries representing the logical structure of the specified file. These dictionaries are utilized by most of the other ADAPT I processes during their processing of various user statements and commands. DDL has its own "front-end" which is constructed to recognize DDL statements; i.e., field definitions, aggregate definitions, etc. From this point on, DDL has control of the user's terminal and will not return to the EXEC process until the user enters an ESCHEMA statement. Since a user may have a UNIX file of DDL statements "executed" (via the EXECUTE command), the EXEC process, which processes the EXECUTE command, establishes the user-specified UNIX file as the "system input" prior to calling DDL. Upon completion of the DDL process, whether DDL has been interacting with the user or reading from a user's UNIX file, control is returned to the EXEC process.

SCENARIO 2 – Assume at this point a user has successfully logged-on to ADAPT I and is now interacting with the EXEC process. The user now wishes to query "file-x" which happens to reside on an "interactive" distant host. The "front-end" of EXEC checks the syntactic validity of the entered FIND statement and, if the statement is valid syntactically, calls the VALIDATE process for semantic validation. VALIDATE performs two basic functions:

a. It verifies that the user's statement, though lexically and syntactically correct, is also compatible with the file being referenced. This involves verifying that the proper relational operators are being used against the correct data types, the user isn't violating the data structures' access or display attributes, and the data structures as named do indeed exist for the specified file.

b. The internal representation of the user's statement (a parse tree) is modified to reflect more discerning information than was possible during the parsing process performed by the EXEC "front-end."

Upon successful completion of the VALIDATE process, EXEC calls the Target System Input Generator (TSIGI, the interactive version) for the actual execution of the FIND statement. Remember, the user is querying a file that resides on an "interactive" host system. TSIGI is responsible for performing the transformation of UDL statements into the appropriate distant host query language. Transformation strings are generated by TSIGI and are sent through the network directly via a NCP daemon system call (a "write"). TSIGI then waits for a response (assuming the distant host is indeed available and an initial connection was successful). Normally the response will be the distant host's acknowledgement of the query and the number of records satisfying it. This information is output to the user as a UDL record hit count. Then TSIGI returns to EXEC which, in turn, prompts the user for his next statement or command. Although in practice a user will normally want to display some portion of the records he has selected, he may also wish to query another file (which may exist on an entirely different distant host system). File-host system association is known by ADAPT I and, therefore, the user does not have to be concerned with the actual system the file resides on.

Assume now that the user is satisfied with the results of the FIND statement just entered and desires to display a set of fields from the selected record set. This is accomplished by entering a DISPLAY statement which references the previously input FIND statement (by a statement label). Upon successful entry of the DISPLAY statement into EXEC, the VALIDATE process is again called to check the semantic validity of the user statement. Assuming compliance in these areas, the EXEC process calls TSIGI again for further processing. As was the case with the FIND statement, TSIGI generates the appropriate transformation strings and transfers them across the network to the respective distant host. TSIGI waits for the host responses which normally will be a set of record data formatted in a manner peculiar to that distant host system. These responses are converted to logical response strings by TSIGI, after which the Target System Output Translation (TSOT) process is called for further analysis. TSOT is responsible for translating the various host system record outputs into compatible UDL internal record formats.

TSOT has the difficult task of distinguishing host error messages from host record data and converting the host data into a set of UDL internal records. If an error message is encountered from the host system, TSOT translates the error message into a consistent UDL error message and outputs it to the user. Assuming successful completion of the TSOT process, EXEC now calls the DISPLAY process to process the user's DISPLAY statement. At this point, the peculiarities of the different distant host systems have disappeared entirely, the DISPLAY process expecting and processing only data that are in the UDL record format. DISPLAY extracts the data as specified in the DISPLAY statement, outputs these data in UDL default format, and then returns to the executive, EXEC.

SCENARIO 3 – In the previous scenario, the user interrogated a file that happened to exist on an "interactive" host system. In this scenario, the user interrogates a file which resides on a "batch" host system. These distinctions are brought out to illustrate the inherent differences in processing techniques that ADAPT I must go through when an "interactive" file is queried as compared to a "batch" file.

To accommodate these differences, two TSIG processes exist in ADAPT I: TSIGI for interactive file references and TSIGB for batch file references. Interestingly, ADAPT I does not require two "TSIG" processes because of differences in the host system query languages. The ADAPT I decompiler, which controls the actual generation of the appropriate transformation strings, is used by both "TSIG" processes. The reason for having two "TSIG" processes is due to the inherent operational differences between interactive processing and transaction-oriented batch processing. That is, the difficulty does not lie with the query languages per se, but with the environment in which they operate in their respective host systems.

Operationally within ADAPT I, the user who references an interactive file is actually "connected" to the distant interactive host system. The ADAPT I system can be thought of as some kind of bidirectional filter absorbing UDL as user input and generating UDL displays as output. For batch files, the user UDL statements are composed into logical transactions which are then sent to the respective batch system for processing. The user is not "connected" to this system and therefore can initiate other queries against other files (interactive or batch). The responses which eventually result from the transaction are processed and saved for the user (under a logical transaction number) for later perusal.

Assume as in the previous scenario that a user has successfully entered a FIND statement referencing some file which resides on a batch system. Also assume that the statement has passed semantic validation. At this point, EXEC calls the Target System Input Generator for batch file references (TSIGB process). TSIGB reserves a logical transaction number for a potential transaction and then generates the appropriate transformation strings representing the FIND statement for that host system. It then returns to EXEC. Note that there was no network communication at this time. For a logical transaction to be complete, it must do more than just interrogate a file. It must ask for some data contingent upon the interrogation as well. This can be accomplished in UDL by either requesting that the selected records be saved in a local list or by displaying some portion of the record set. Notice that there is no true interaction with the user and distant host system. The user must package his transaction as described in the foregoing and then wait for the results. Upon successful entry of, say, a SAVE command, EXEC calls TSIGB again (assuming the SAVE command did reference the previous FIND statement) for further processing. TSIGB performs the appropriate transformation on the SAVE command and generates a set of transformation strings which then are added to the end of the previous strings generated by the previous FIND statement. At this point, a logical transaction can be packaged and sent through the network. To do this, TSIGB must call the Batch Network Control Program (BNCP, also known as the "Batch Query Processor") which will generate the proper COINS I INTG message and transmit it through the network via the NCP daemon. TSIGB then outputs the logical transaction number to the user and returns to the executive, EXEC. At this point, the user is prompted and can now enter new statements or commands.

Upon receipt of a COINS I ANSR or ABRT message (and assuming it is a message pertaining to the logical transaction initiated in the foregoing), BNCP will call the Network Response process (NETRES) of ADAPT I for processing. At the time when TSIGB established the logical transaction through BNCP, it also specified that NETRES be called when a response for this transaction occurred.

NETRES analyzes the message received from BNCP. If the response was an ANSR, which would normally be the case, TSOT is called to perform its function of translating host system record data into proper UDL records. Upon the completion of TSOT, NETRES determines if the user requested the display of data. In this particular scenario, the user was only interested in saving the selected records in an internal list local to him. Therefore, NETRES indicates that the logical transaction is now complete and exits. If the user had requested the display of data, NETRES would have called the DISPLAY process which would display the data in UDL default format onto a temporary output file associated with this particular logical transaction. At a later time, the user can peruse this output by requesting it through the VIEW command.

SCENARIO 4 – If a user desires to see the logical structure of some file, he may do so by entering the VIEW command. The EXEC process calls the VALIDATE process to verify the semantic correctness of the command and then calls the VIEW process to output the file's logical structure. VIEW not only outputs file structures, it can also display the names of files currently defined under ADAPT I, the names of lists defined locally by the requesting user, and all pending and/or completed batch transactions initiated by that user.

If users wish to logoff of ADAPT I, they must enter the LOGOFF command. EXEC calls the COMMAND process which removes all local statement labels existing for that user, indicates that the user is "inactive," and returns to the executive. EXEC then returns to the UNIX shell.

ADAPT I PROCESS DESCRIPTIONS

This section provides a fairly detailed description of the 11 processes currently envisioned for ADAPT I. The 11 processes described in this section are as follows:

a. ADAPT – processes initial user logon to ADAPT I.

b. EXEC – performs overall executive operations for ADAPT I including UDL statement/command syntactic analysis, and the transferring of control to appropriate ADAPT I processes for statement/command processing.

c. VALIDATE – performs semantic validation of UDL statements and commands.

d. COMMAND – processes certain UDL commands.

e. VIEW – performs all operations involving the display of SCHEMA structures, user list-names, and user transactions.

f. DDL – processes all DDL statements and generates file dictionary data files.

g. TSIGI – generates transformation strings for queries of files residing on interactive host systems.

h. TSIGB – generates transformation strings for queries of files residing on batch host systems.

i. NETRES – processes batch host system responses and maintains the user transaction data file.

j. TSOT – translates distant host system responses into UDL internal data records.

k. DISPLAY – processes the user's DISPLAY statement, outputting UDL default formatted record data.

Each process description is organized as follows. A short general description is provided. This is followed by a list of the ADAPT I global data used, accompanied by a short description of how it is used. Then, if applicable, a list of pertinent "local" data contained in the process is provided. A general data flow is then given which narrates the overall paths the process must take in order to accomplish its tasks. After the general flow, salient functions contained in the process are highlighted. These functions, which were called out by name in the general flow description, are described in more detail. Following the process description are flow charts which illustrate the actual program paths of the process. Again, as with descriptions, salient functions have flow charts provided for them.

ADAPT (ADAPT Logon Process)

The ADAPT process is activated when a user enters the ADAPT command. ADAPT validates the user's name and activates the ADAPT executive, EXEC.

GLOBAL DATA USAGE – ADAPT utilizes GUSER – user descriptions.

LOCAL DATA USAGE – ADAPT has no pertinent local data.

GENERAL PROCESS FLOW – ADAPT is called by the UNIX shell program when a user types in ADAPT and a user name. If no name argument is input, a diagnostic is output and ADAPT terminates. Otherwise, ADAPT locks the user descriptions (GUSER) via GFLOCK and reads GUSER. ADAPT validates that the input user name is a valid user name and that the user is not already logged-on at another terminal. If an error condition exists, a diagnostic is output via GFERROR, ADAPT unlocks GUSER via GFUNLOCK, ADAPT terminates, and the user is back at the shell program level. If no error is found, ADAPT sets the user's status to logged-on, writes GUSER, and unlocks GUSER via GFUNLOCK. It then passes control to the ADAPT executive, EXEC. See figure 34 for data flow.

EXEC (ADAPT Executive)

The EXEC process initializes and controls the ADAPT environment for users while they are logged-on. As each user types in an ADAPT statement or command, EXEC parses this input and builds a parse tree. EXEC then has the statement validated, if applicable, and either processes the statement or calls the appropriate processor.

GLOBAL DATA USAGE – EXEC uses the following global data:

GTREE – parse tree.
GTDAT – parse tree data.

LOCAL DATA USAGE – EXEC uses the following local data:

LXKYWRD – keywords.
LXTOKEN – keyword token types.

GENERAL PROCESS FLOW – EXEC is activated whenever a user successfully logs-on to ADAPT and remains in control until the user logs-off. EXEC calls the LEX/parse initialization portion of the process (LXINIT) which causes the next input statement to be read in and parsed and a parse tree to be built. If a syntax error is encountered in the statement, an error message is output and EXEC goes on to the next statement. Otherwise, the statement's primary token type is picked up and EXEC determines whether the statement needs semantical validation. If so, the VALIDATE process is called to validate and modify the statement parse tree. If there was a semantic error in the statement, EXEC goes on to the next statement.

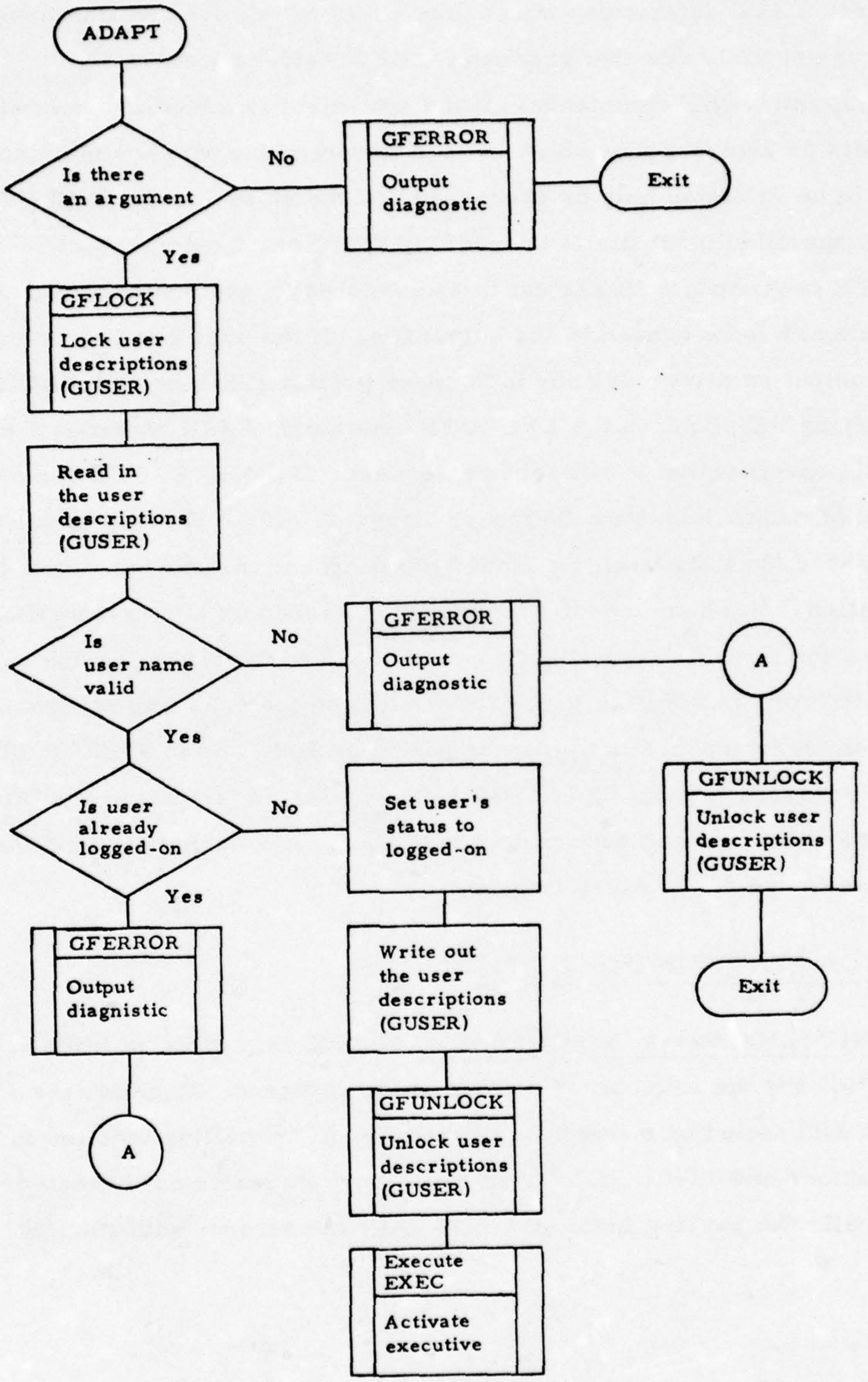

ADAPT
Is there an argument
No
GFERROR
Output diagnostic
Exit
Yes
GFLOCK
Lock user descriptions (GUSER)
Read in the user descriptions (GUSER)
Is user name valid
No
GFERROR
Output diagnostic
A
Yes
Is user already logged-on
No
Set user's status to logged-on
GFUNLOCK
Unlock user descriptions (GUSER)
Yes
GFERROR
Output diagnistic
Write out the user descriptions (GUSER)
Exit
A
GFUNLOCK
Unlock user descriptions (GUSER)
Execute EXEC
Activate executive

Figure 34. ADAPT Process Data Flow.

Otherwise, EXEC determines which process is to receive the statement for processing and calls that process. EXEC itself processes the EXECUTE and MODE commands. If the statement is a MODE command, EXEC sets or clears a flag which is used to determine whether the statement is to be validated only or processed. If the statement is EXECUTE, the user-specified input file is opened. If ECHO has been specified on the EXECUTE command, a flag is set to specify that all statements from the user's file are to be echoed to the output file. If the user has requested that the output be directed to the high speed printer (HSP) or to a UNIX file by specifying REMOTE on the EXECUTE command, EXEC modifies the output file specification to reflect this request. If the EXECUTE command specifies SCHEMA, the Data Definition Language (DDL) process is called and is passed the echo flag, the input file specification, and the output file specification. When an end-of-file (EOF) is reached on a user-specified input file, the input file specification is set back to the terminal, the output file specification is set back to the terminal, and the echo flag is cleared. If a system error occurs in a process called by EXEC or in EXEC itself, an error message is output and ADAPT is terminated for that user. After a LOGOFF command has been processed, EXEC also terminates for that user. See figure 35 for more information.

MAJOR FUNCTION DESCRIPTIONS –

LXINIT (LEX/Parse Initialization) – LXINIT is merely an initialization function for the LEX/parse portion of this process. It initializes pertinent data including a flag which it returns to the calling function to indicate either end-of-file (EOF), syntax error, or statement accepted. LXINIT calls the parsing function (YYPARSE) and returns with the flag value.

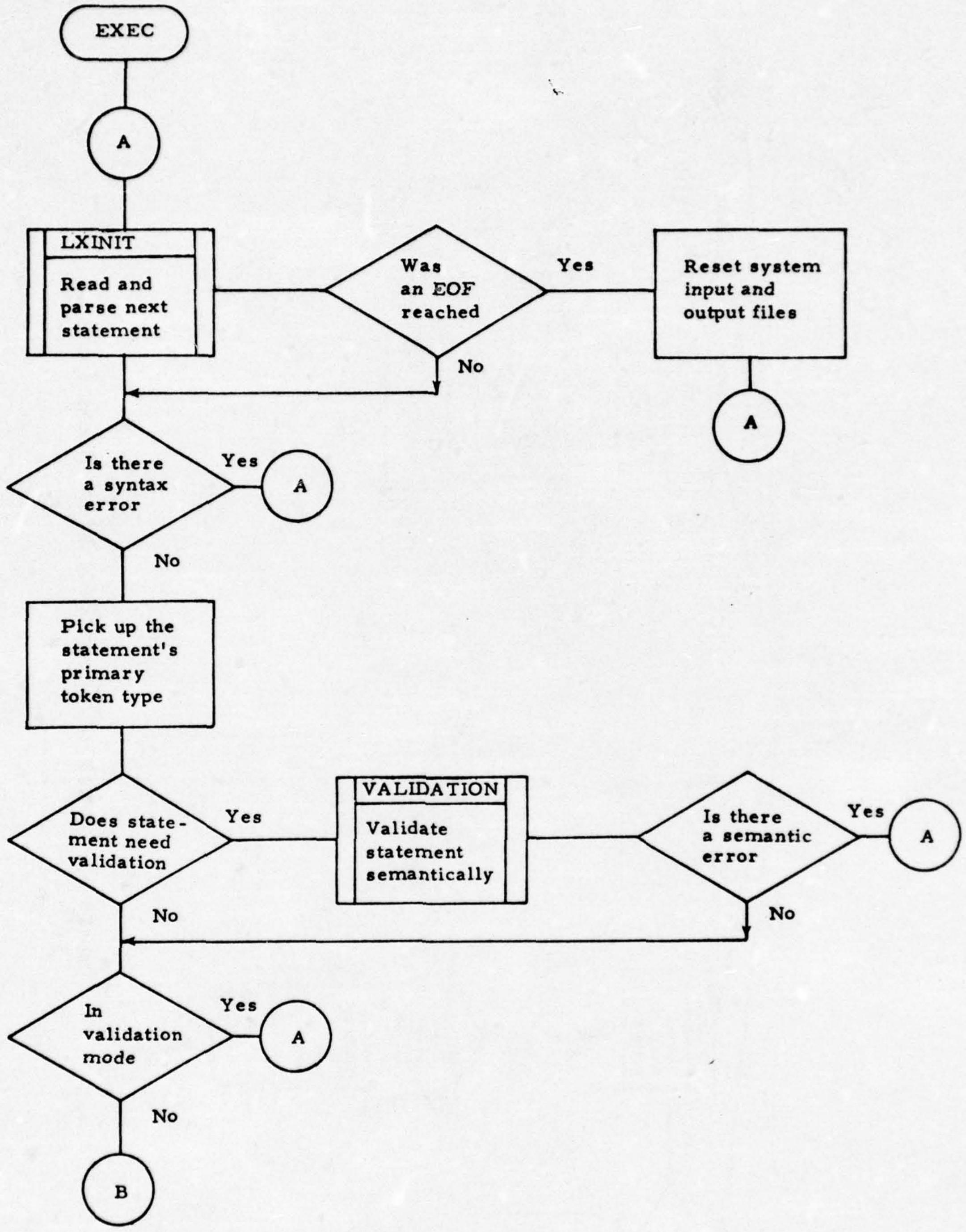

EXEC
A
LXINIT
Read and parse next statement
Was an EOF reached
Yes
Reset system input and output files
A
No
Is there a syntax error
Yes
A
No
Pick up the statement's primary token type
Does statement need validation
Yes
VALIDATION
Validate statement semantically
Is there a semantic error
Yes
A
No
No
In validation mode
Yes
A
No
B

Figure 35. EXEC Process Data Flow (Sheet 1 of 6).

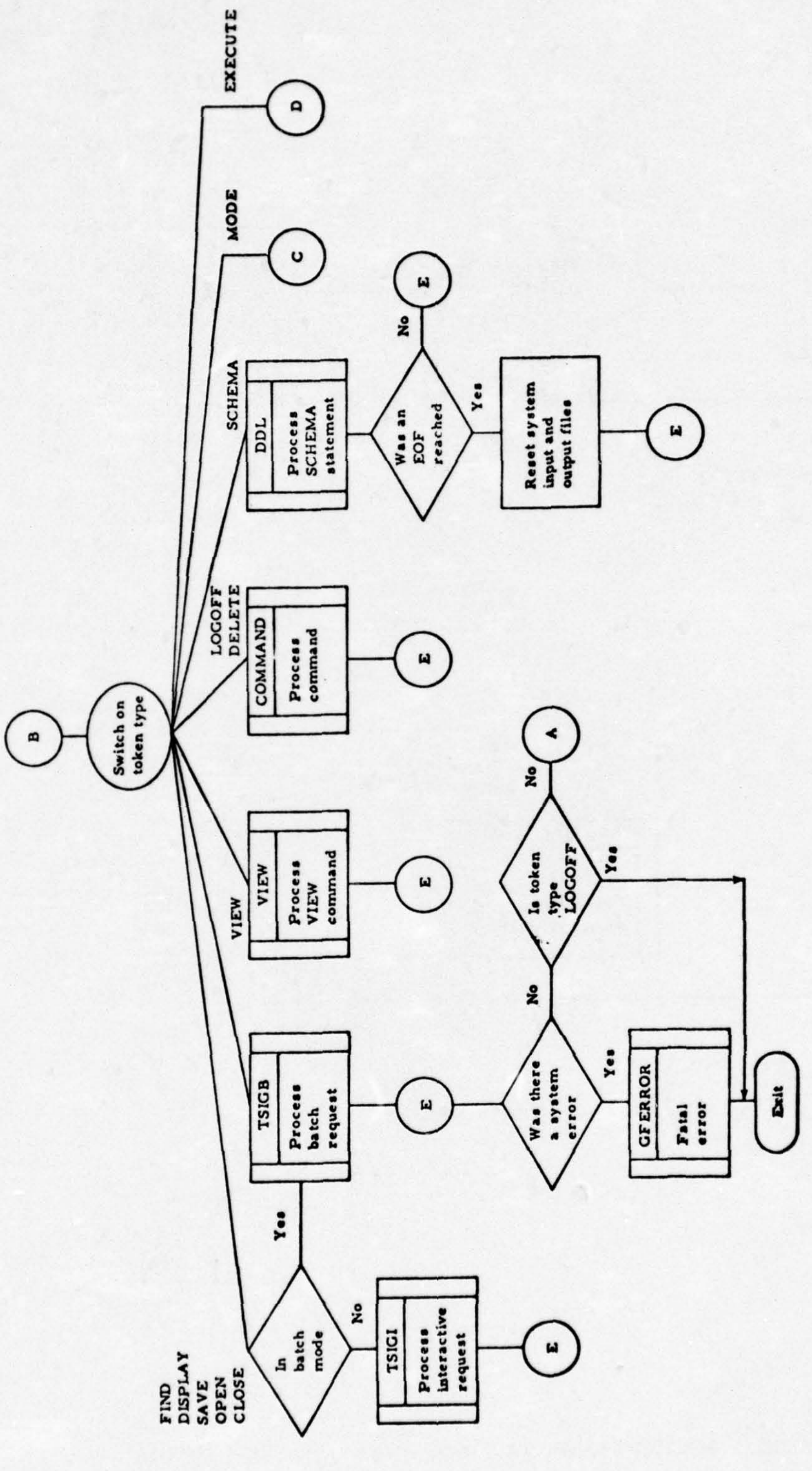


Figure 35. EXEC Process Data Flow (Sheet 2 of 6).

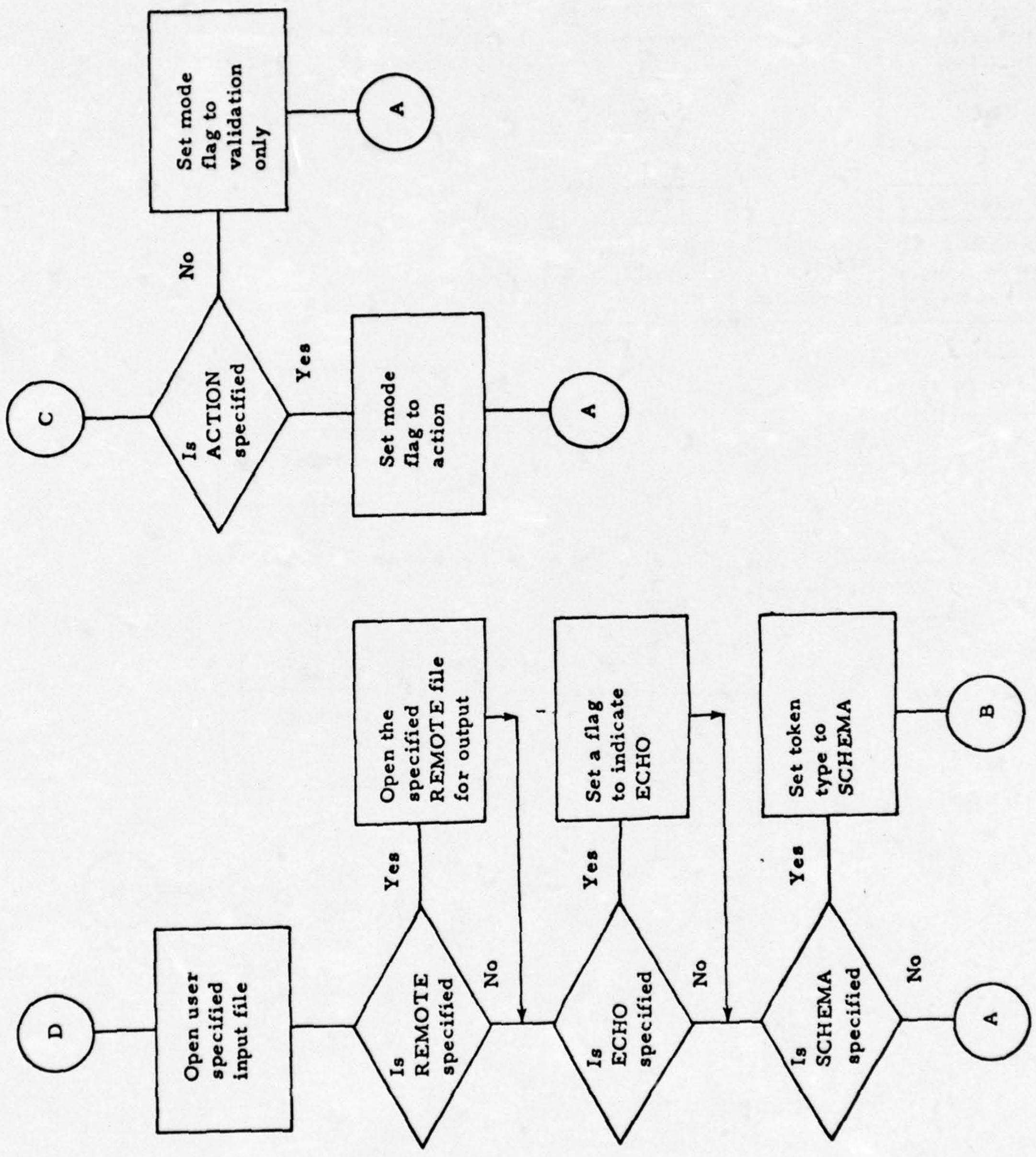


Figure 35. EXEC Process Data Flow (Sheet 3 of 6).

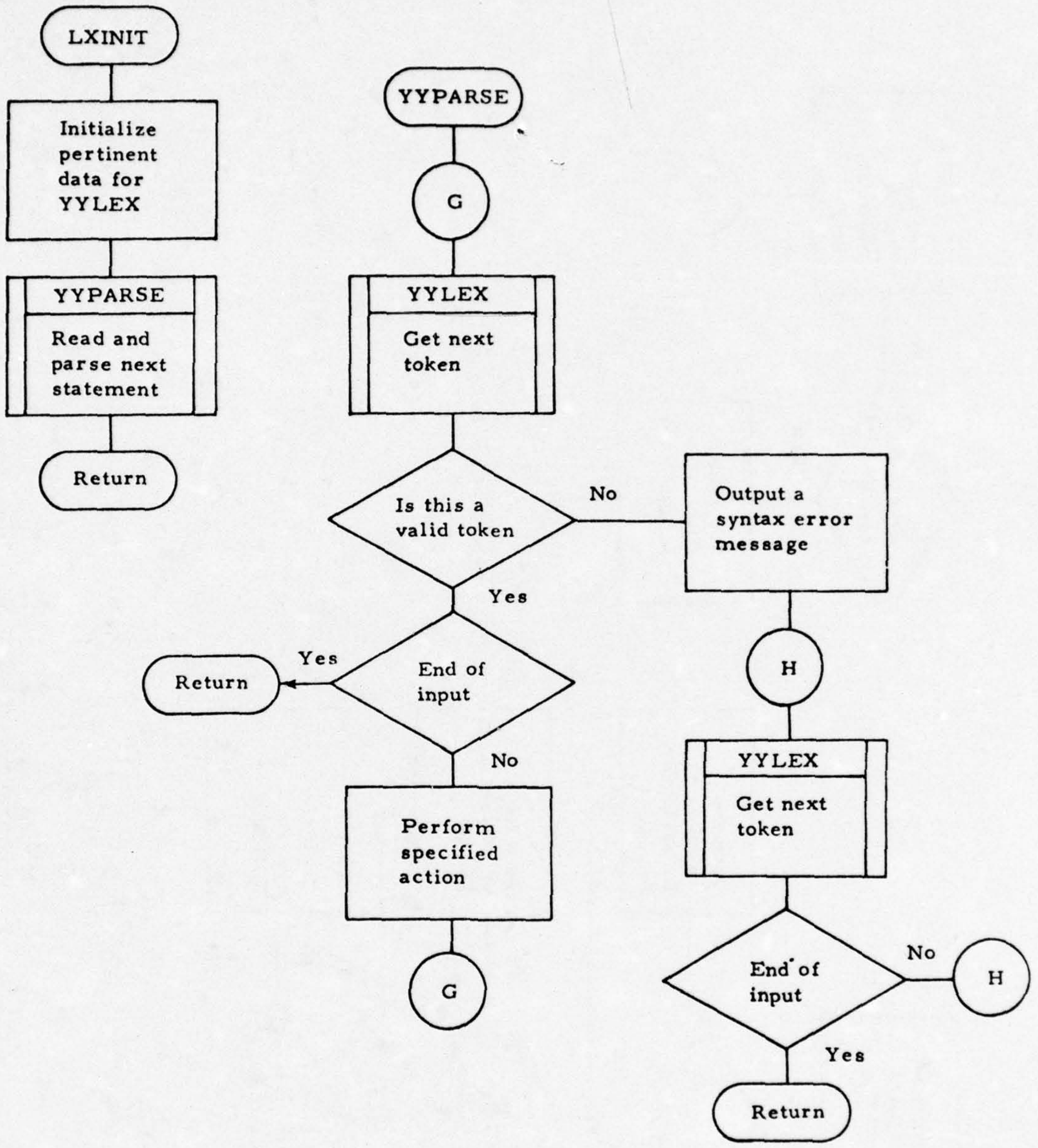


Figure 35. EXEC Process Data Flow (Sheet 4 of 6).

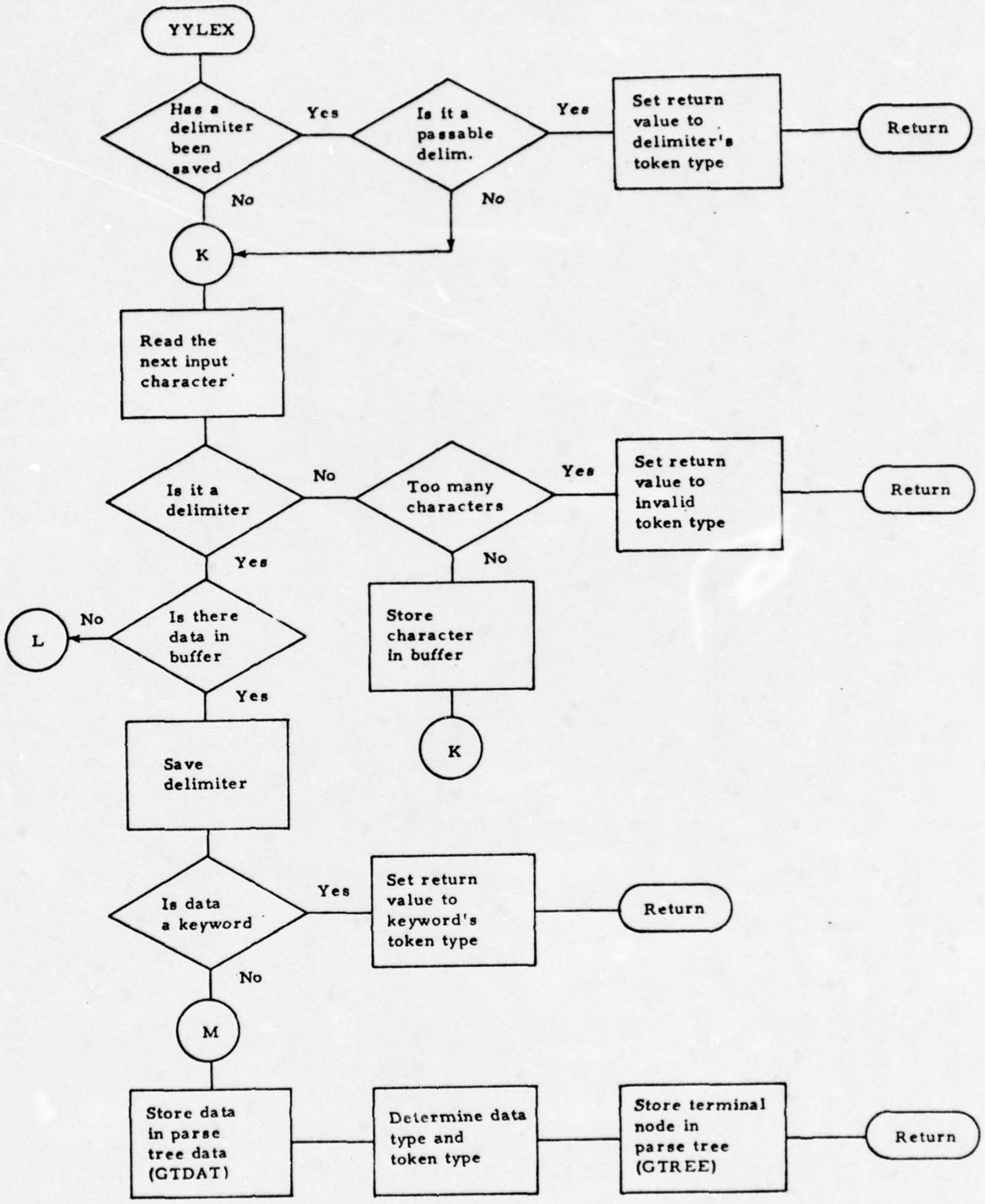

YYLEX
Has a delimiter been saved
Yes
Is it a passable delim.
Yes
Set return value to delimiter's token type
Return
No
No
K
Read the next input character
Is it a delimiter
No
Too many characters
Yes
Set return value to invalid token type
Return
Yes
No
No
Is there data in buffer
L
Store character in buffer
K
Yes
Save delimiter
Is data a keyword
Yes
Set return value to keyword's token type
Return
No
M
Store data in parse tree data (GTDAT)
Determine data type and token type
Store terminal node in parse tree (GTREE)
Return

Figure 35. EXEC Process Data Flow (Sheet 5 of 6).

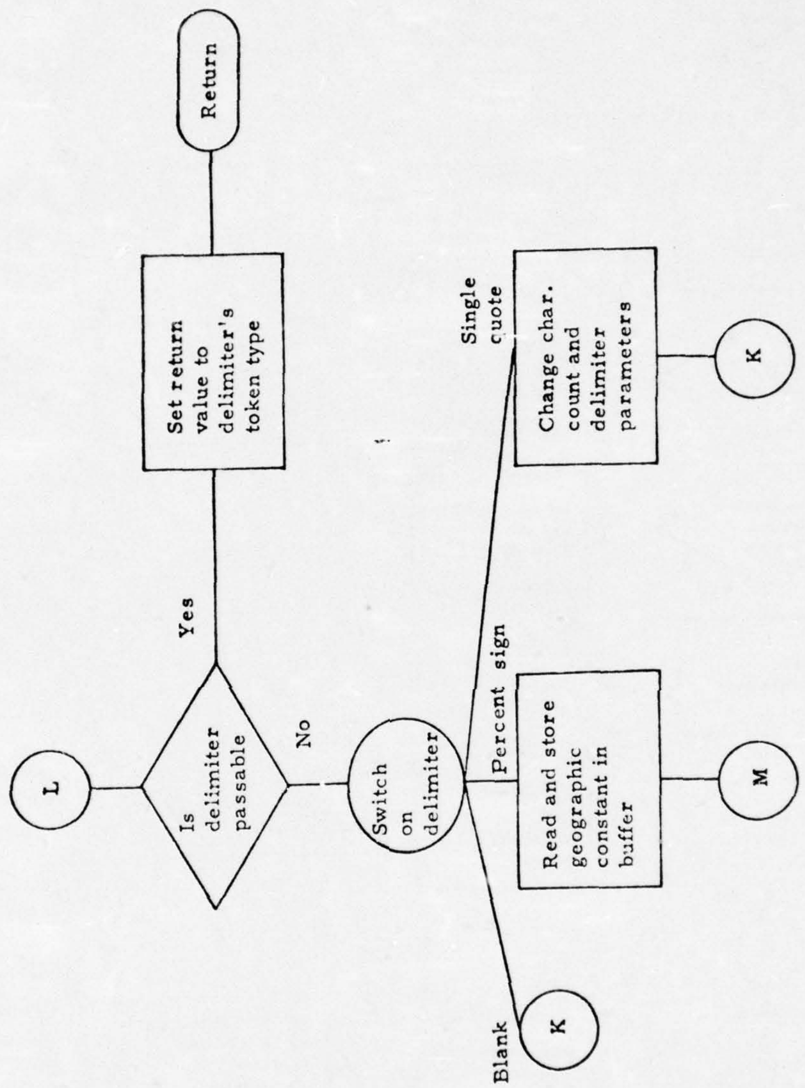

L
Is delimiter passable
Yes
Set return value to delimiter's token type
Return
No
Switch on delimiter
Blank
K
Percent sign
Read and store geographic constant in buffer
M
Single quote
Change char. count and delimiter parameters
K

Figure 35. EXEC Process Data Flow (Sheet 6 of 6).

YYPARSE (Parser) – YYPARSE is a parser which is produced by the Yet Another Compiler-Compiler (YACC) program running under UNIX. YACC produces the parser as well as a set of tables which the parser uses to organize the tokens passed to it by the lexical analyzer (YYLEX). These tables reflect the grammer of the various UDL statements. YYPARSE calls YYLEX, which returns a value called a token type. If the token type is invalid according to the input statement syntax rules, an error message is output and YYPARSE calls YYLEX continuously until an end-of-input token type is returned. YYPARSE then returns with the return flag set to reflect a syntax error. If the token type is valid according to the input rules, the action specified in the YACC run is performed. The actions that can be invoked are: no action, build a parse tree node with the specified number of legs, store the final token type and write out the parse tree and parse tree data, start a list, add an element to the current list, and store the list in the parse tree. After the action is performed, YYPARSE calls YYLEX for the next token type. When the end-of-input token type is received and its action performed, YYPARSE returns with the return flag set to reflect statement accepted. If YYLEX encounters an EOF, it returns an end-of-input token type and sets the return flag to reflect EOF.

YYLEX (Lexical Analyzer) – YYLEX is the function which actually inputs and processes each UDL statement. YYLEX reads and saves each input character until a delimiter is encountered. The delimiters for ADAPT I are: comma, newline (carriage return), single quote mark, percent sign, space, left parenthesis, and right parenthesis. If an EOF is encountered, the return flag is set to reflect EOF and an end-of-input token type is returned. If a delimiter is encountered by itself, it is either passed to the parser (comma, left or right parenthesis, or newline), it is ignored, or it causes YYLEX to change its input mode; e.g., single quote mark causes YYLEX to read characters until another single quote mark is input. When a delimiter terminates a string of characters, YYLEX must determine what the characters represent. If the characters form a UDL keyword, YYLEX returns with that keyword's token type. Otherwise, YYLEX determines the type of constant which is formed by the characters (name, integer, numeric constant, geographic constant, or character constant), stores the value of the constant in the parse tree data (GTDAT), stores a terminal node in the parse tree (GTREE), and returns with the constant's token type. If YYLEX encounters an error (e.g., too many characters input), it returns an invalid token type to the parser and thereby generates a syntax error message.

VALIDATE (Semantic Validation Process)

The ADAPT I VALIDATE process checks statements of UDL for semantic accuracy and legality according to information stored in the global data files. Each statement successfully validated by VALIDATE can be further processed with minimal concern of nonsensical statements and meaningless results. Semantical mistakes are located and diagnostic messages printed out so that the user can quickly correct and re-enter the statement. VALIDATE also modifies the GTREE, when applicable, with more meaningful token types and with cross-references to appropriate global data files. See figure 36 for more information.

GLOBAL DATA USAGE – The following ADAPT I global data tables are utilized extensively by the VALIDATE process:

GFILN – global file name table. Used by statements DELETE, OPEN, FIND, VIEW, and CLOSE.

GLIST – global list name and description table. Used by statements SAVE, DELETE, and DISPLAY.

GLABL – global label (TAG) name and description table. Used by statements SAVE, DISPLAY, and FIND.

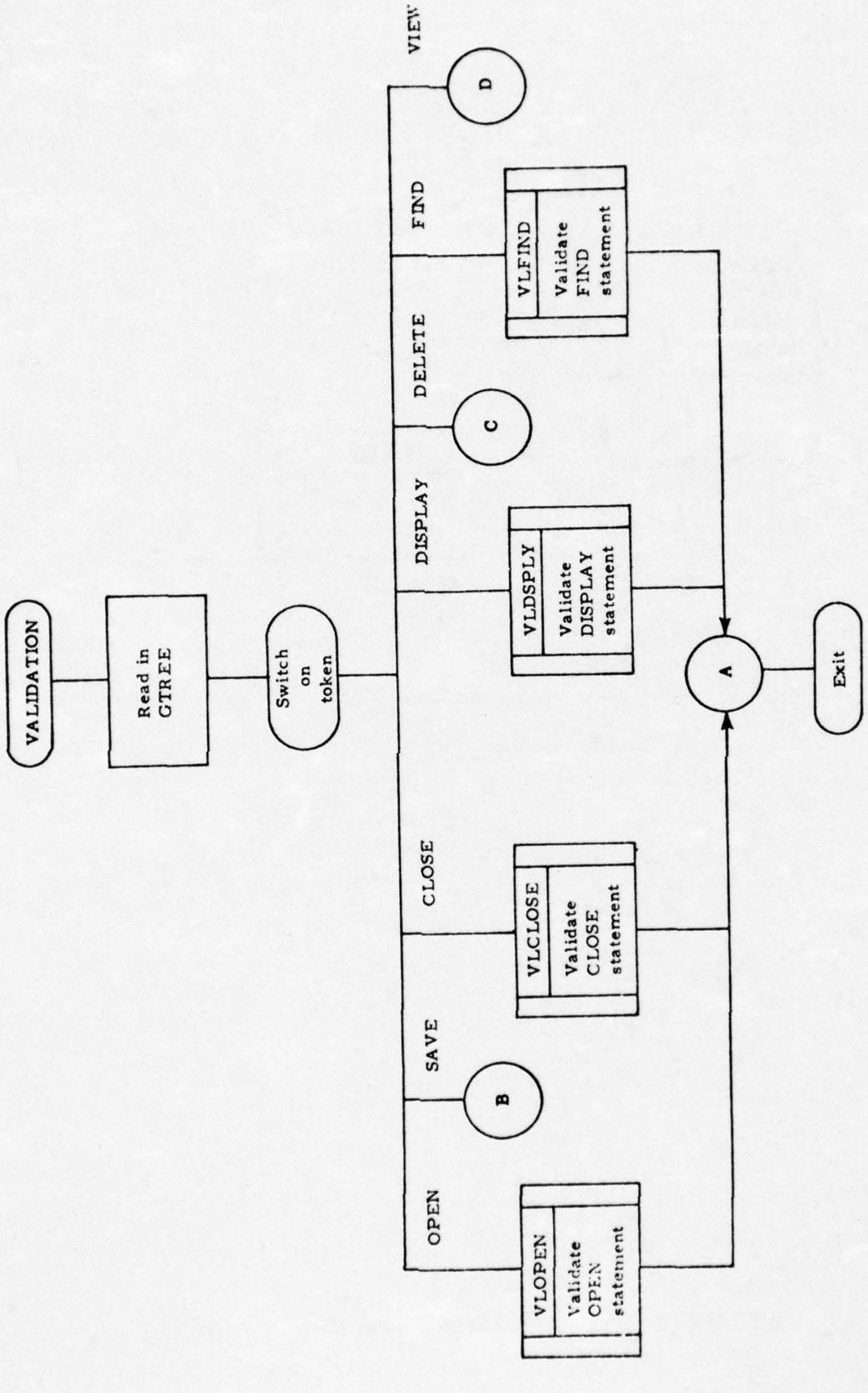


Figure 36. VALIDATE Process Data Flow (Sheet 1 of 16).

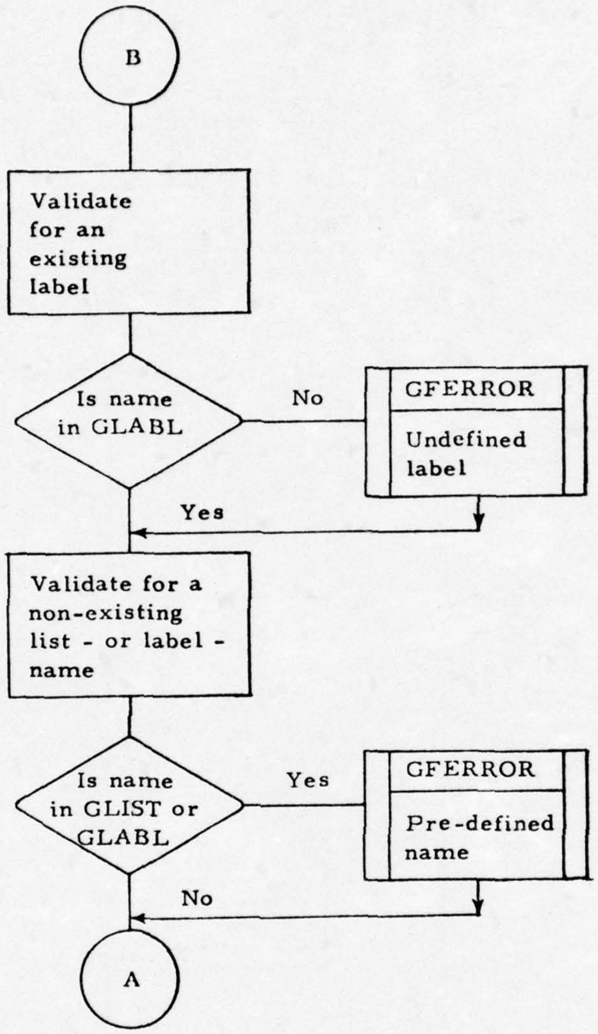

B
Validate for an existing label
Is name in GLABL
No
GFERROR
Undefined label
Yes
Validate for a non-existing list - or label - name
Is name in GLIST or GLABL
Yes
GFERROR
Pre-defined name
No
A

Figure 36. VALIDATE Process Data Flow (Sheet 2 of 16).

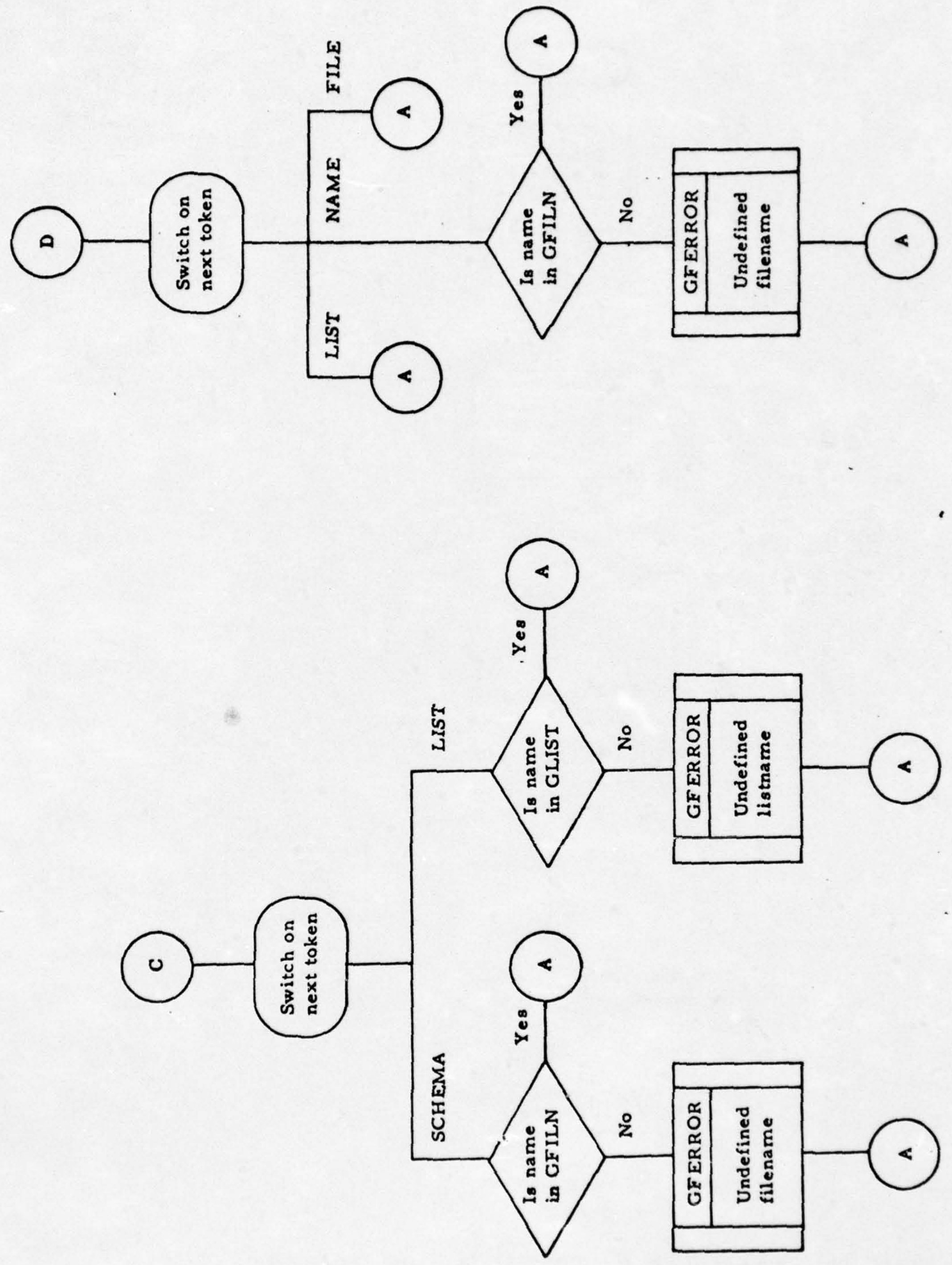


Figure 36. VALIDATE Process Data Flow (Sheet 3 of 16).

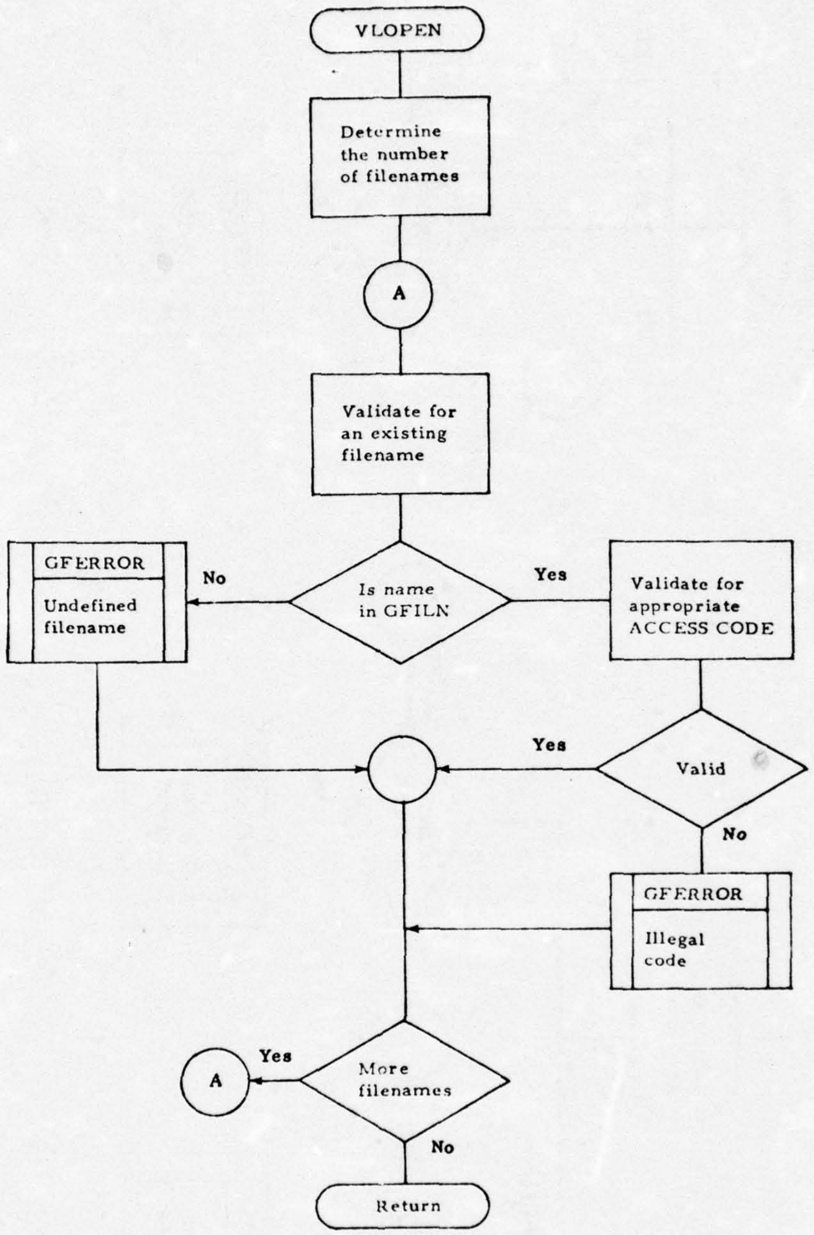


Figure 36. VALIDATE Process Data Flow (Sheet 4 of 16).

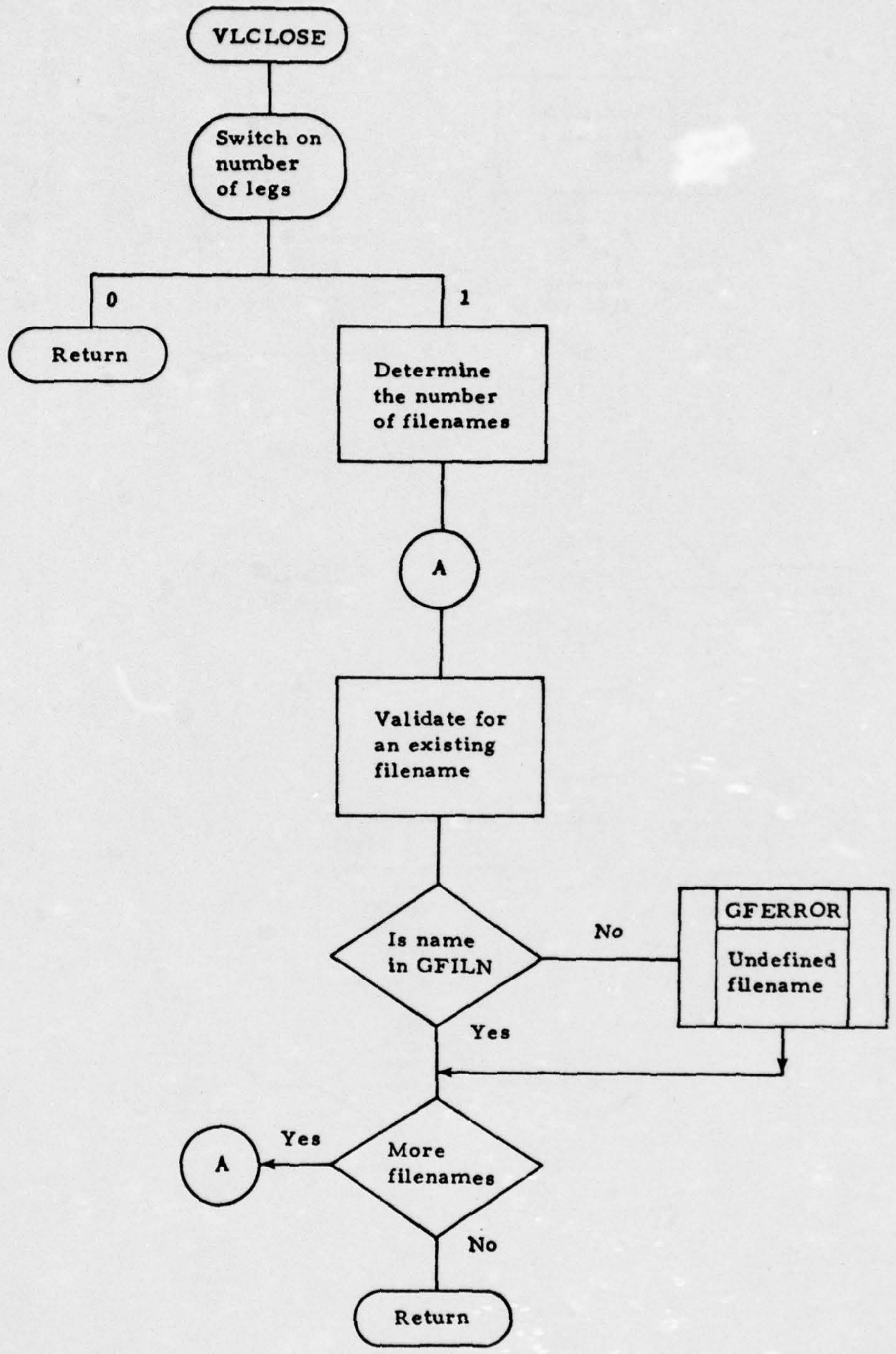


Figure 36. VALIDATE Process Data Flow (Sheet 5 of 16).

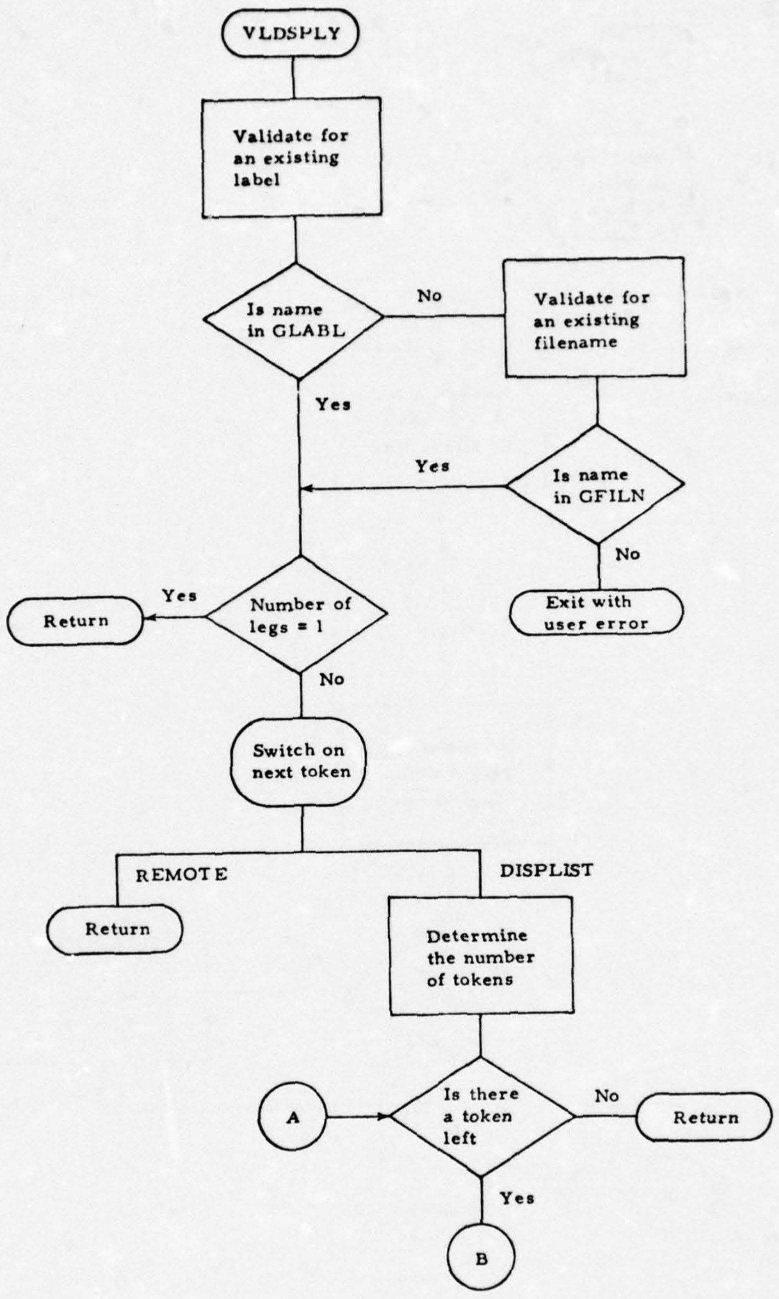


Figure 36. VALIDATE Process Data Flow (Sheet 6 of 16).

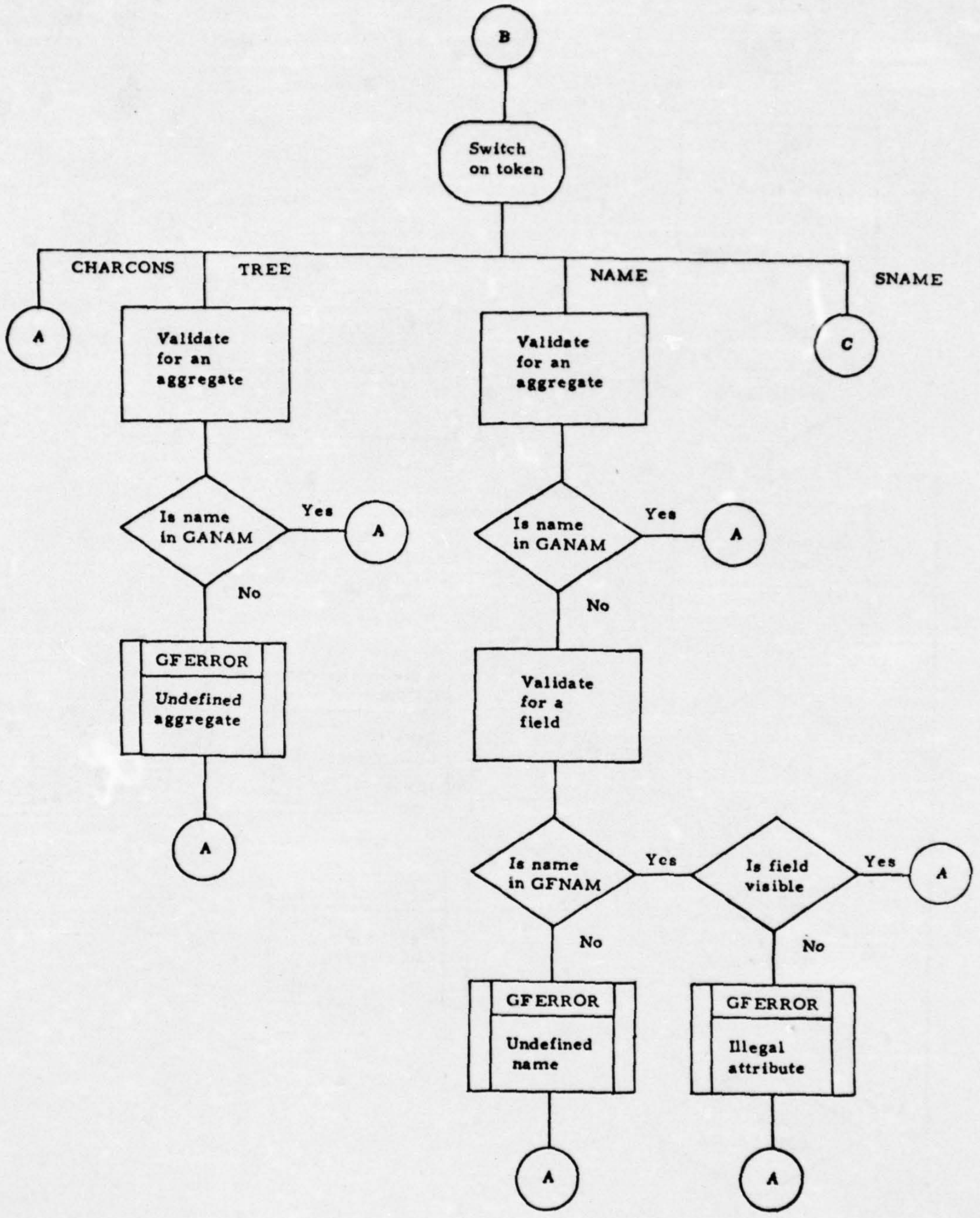


Figure 36. VALIDATE Process Data Flow (Sheet 7 of 16).

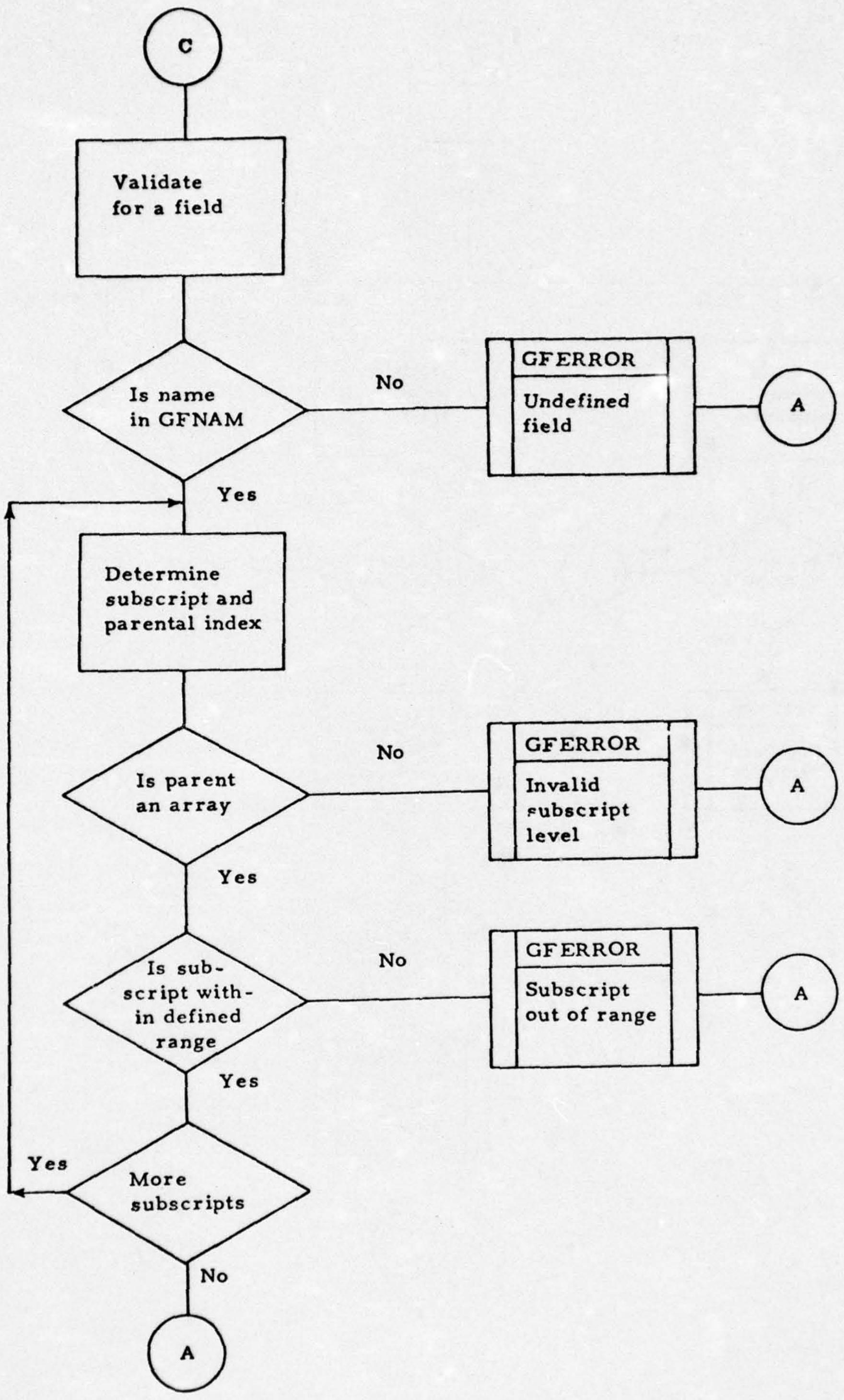


Figure 36. VALIDATE Process Data Flow (Sheet 8 of 16).

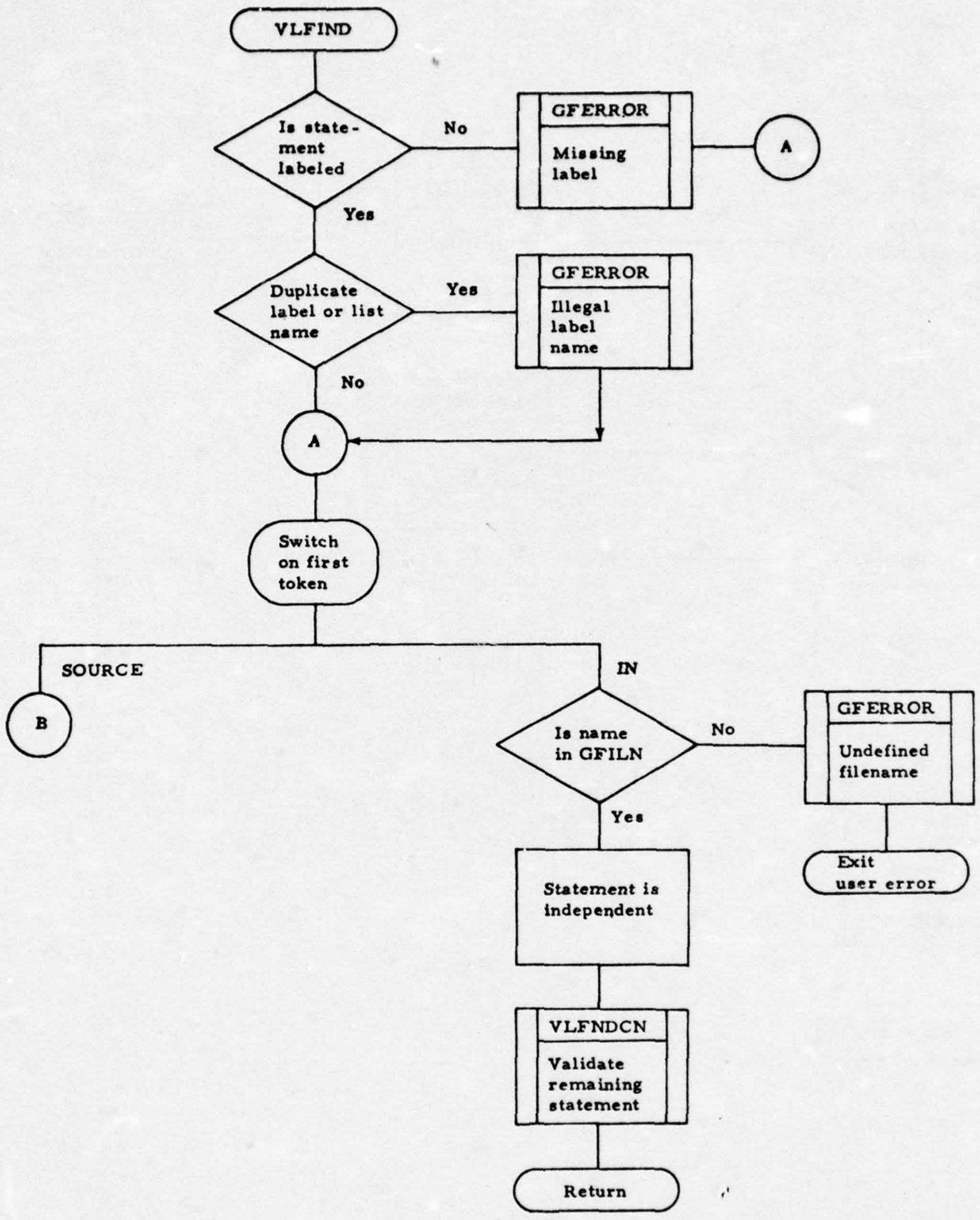


Figure 36. VALIDATE Process Data Flow (Sheet 9 of 16).

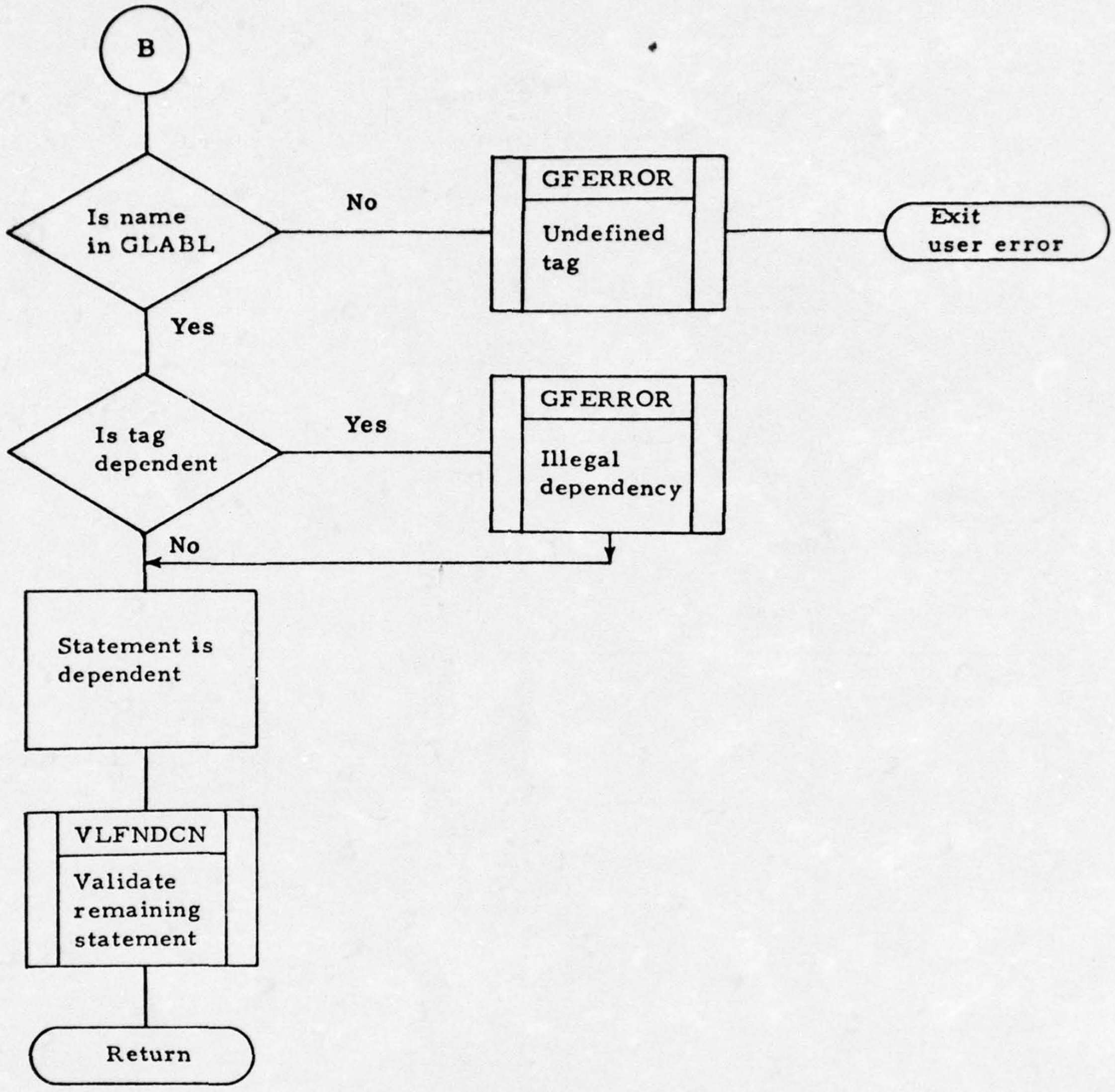


Figure 36. VALIDATE Process Data Flow (Sheet 10 of 16).

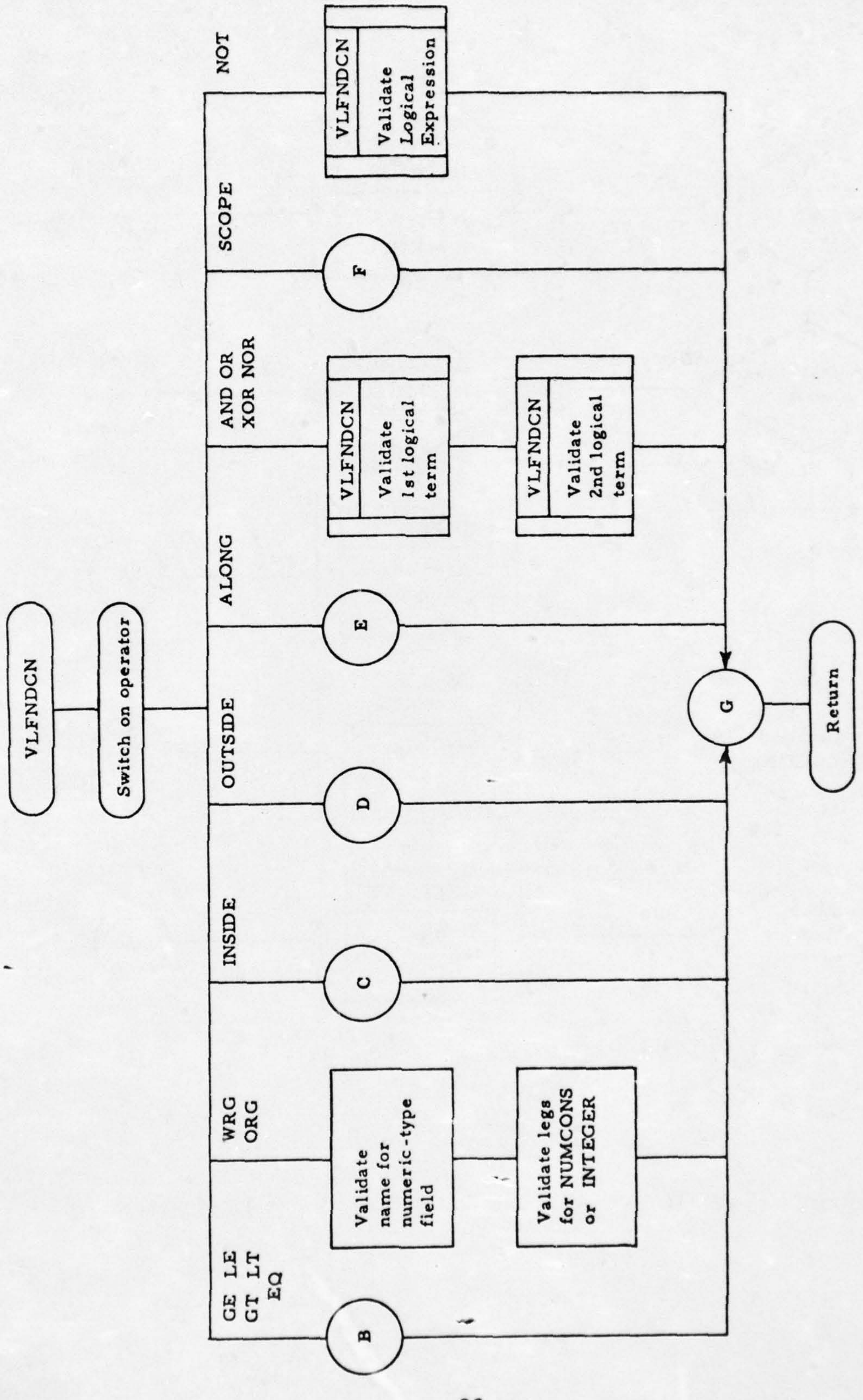

VLFNDCN
Switch on operator
GE LE GT LT EQ
WRG ORG
INSIDE
OUTSIDE
ALONG
AND OR XOR NOR
SCOPE
NOT
B
Validate name for numeric-type field
Validate legs for NUMCONS or INTEGER
C
D
E
VLFNDCN Validate 1st logical term
VLFNDCN Validate 2nd logical term
F
VLFNDCN Validate Logical Expression
G
Return

Figure 36. VALIDATE Process Data Flow (Sheet 11 of 16).

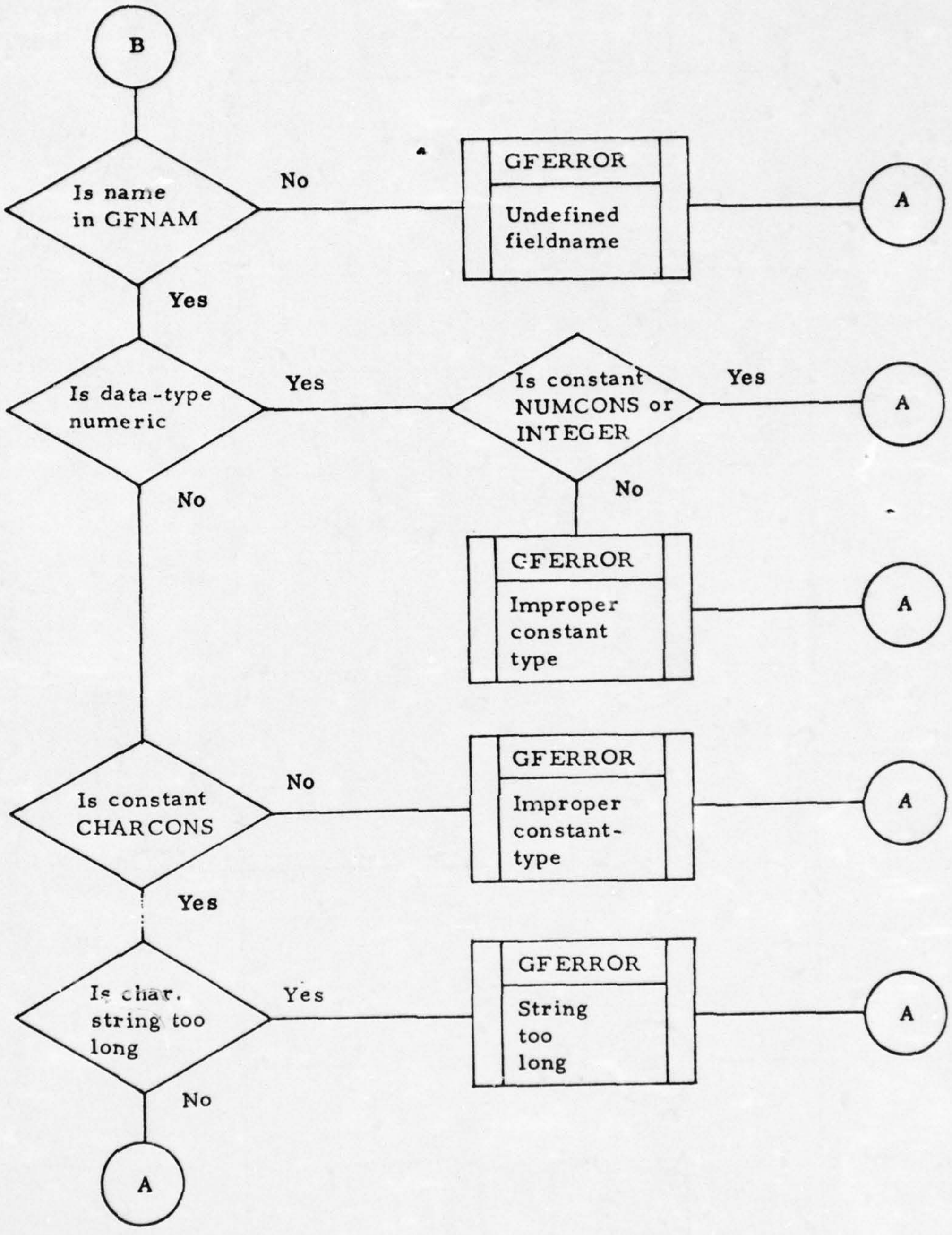
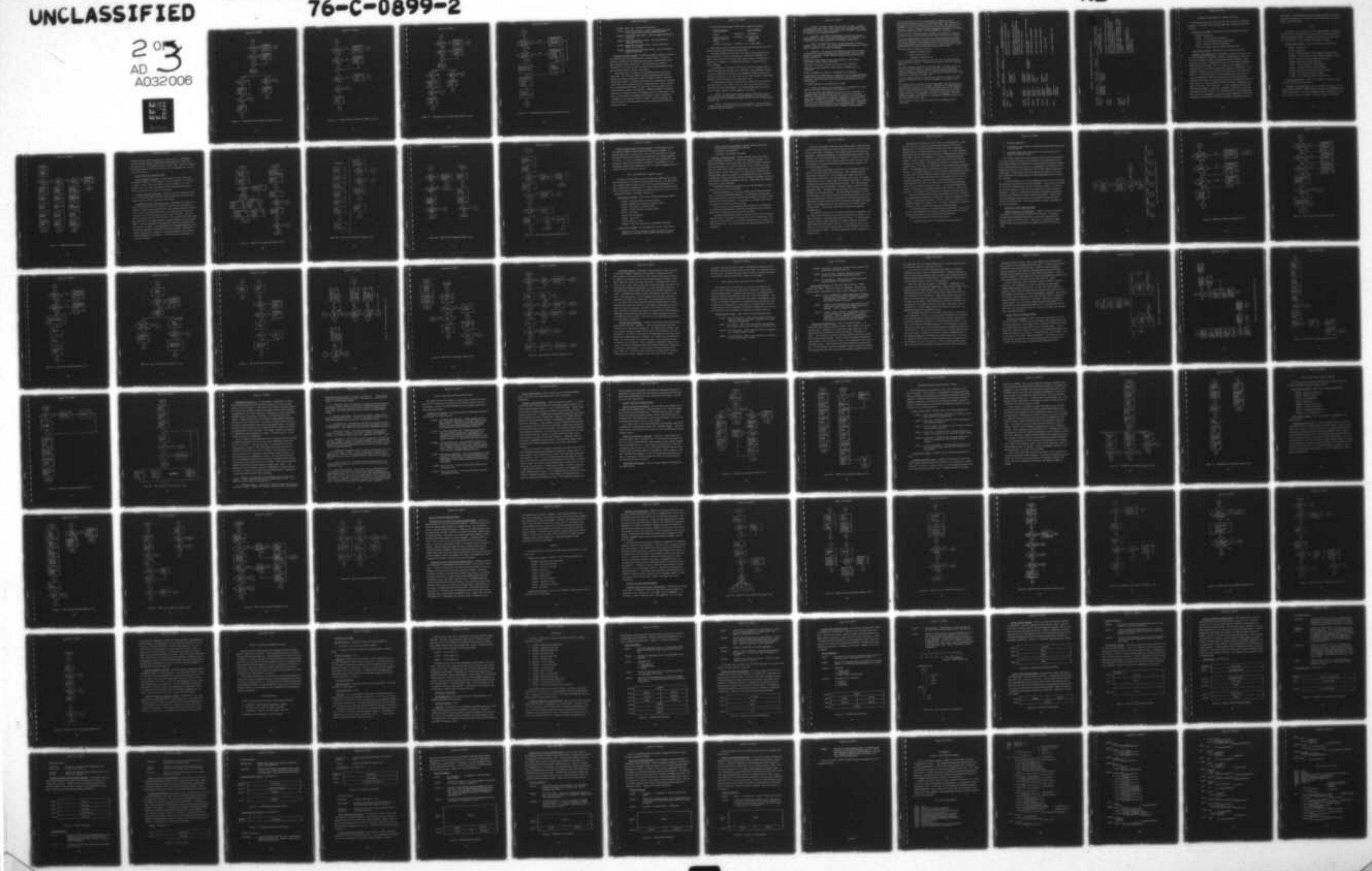


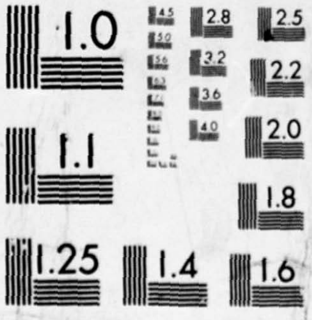
Figure 36. VALIDATE Process Data Flow (Sheet 12 of 16).

AD-A032 006 LOGICON INC SAN DIEGO CALIF F/G 9/2
ADAPT I PRELIMINARY FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION.(U)
NOV 76 L R ERICKSON, M E SOLEGLAD N00014-76-C-0899

UNCLASSIFIED 76-C-0899-2 NL

2 of 3
AD
A032006




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

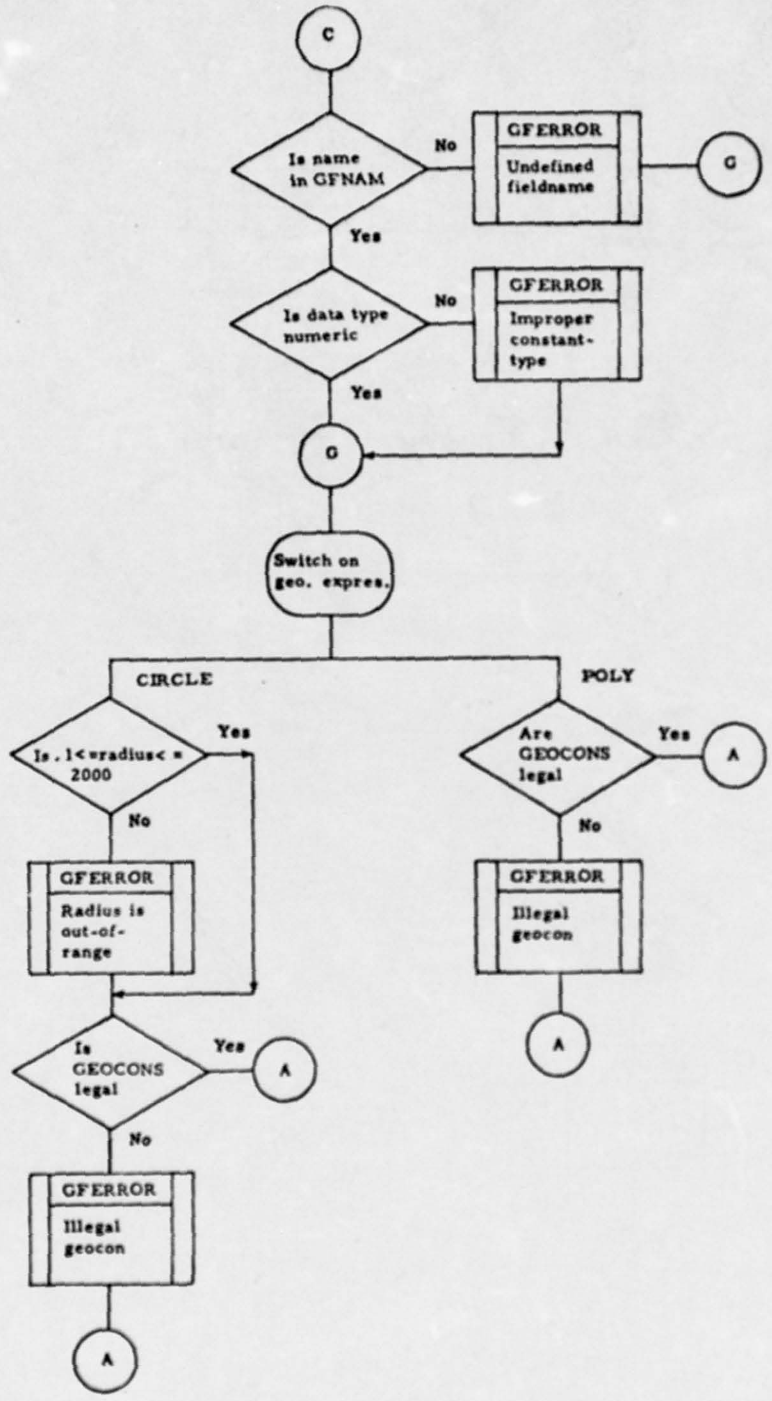


Figure 36. VALIDATE Process Data Flow (Sheet 13 of 16).

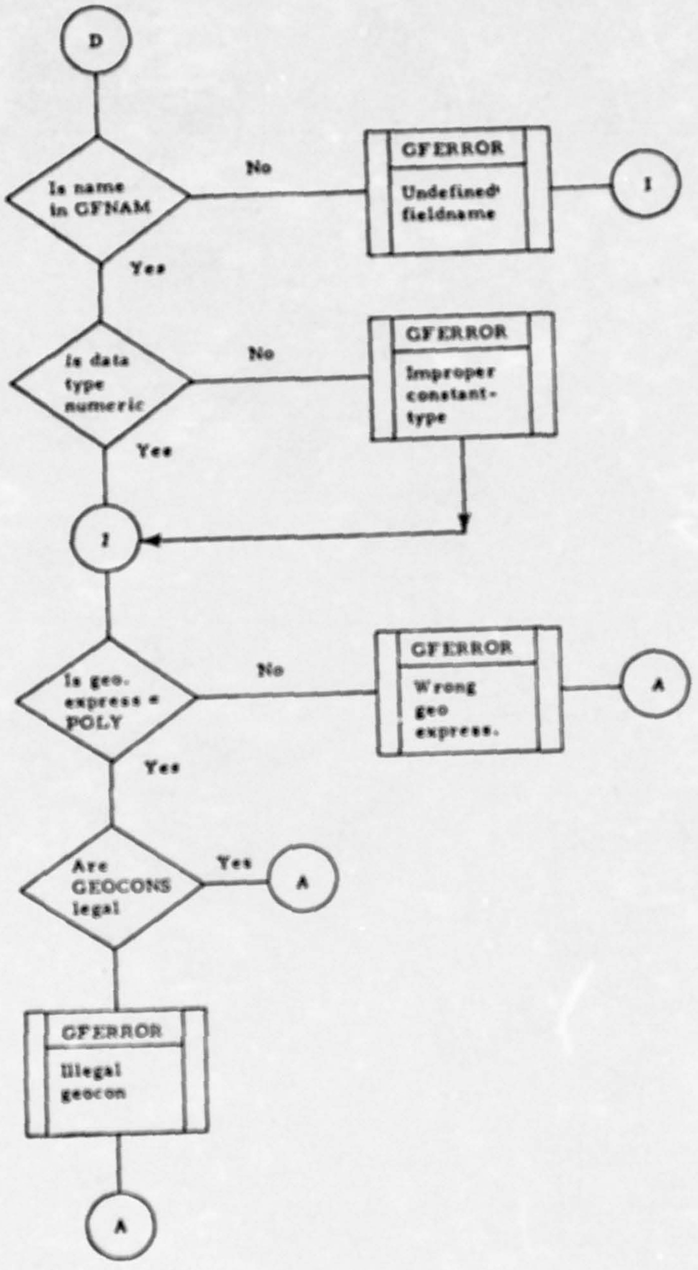


Figure 36. VALIDATE Process Data Flow (Sheet 14 of 16).

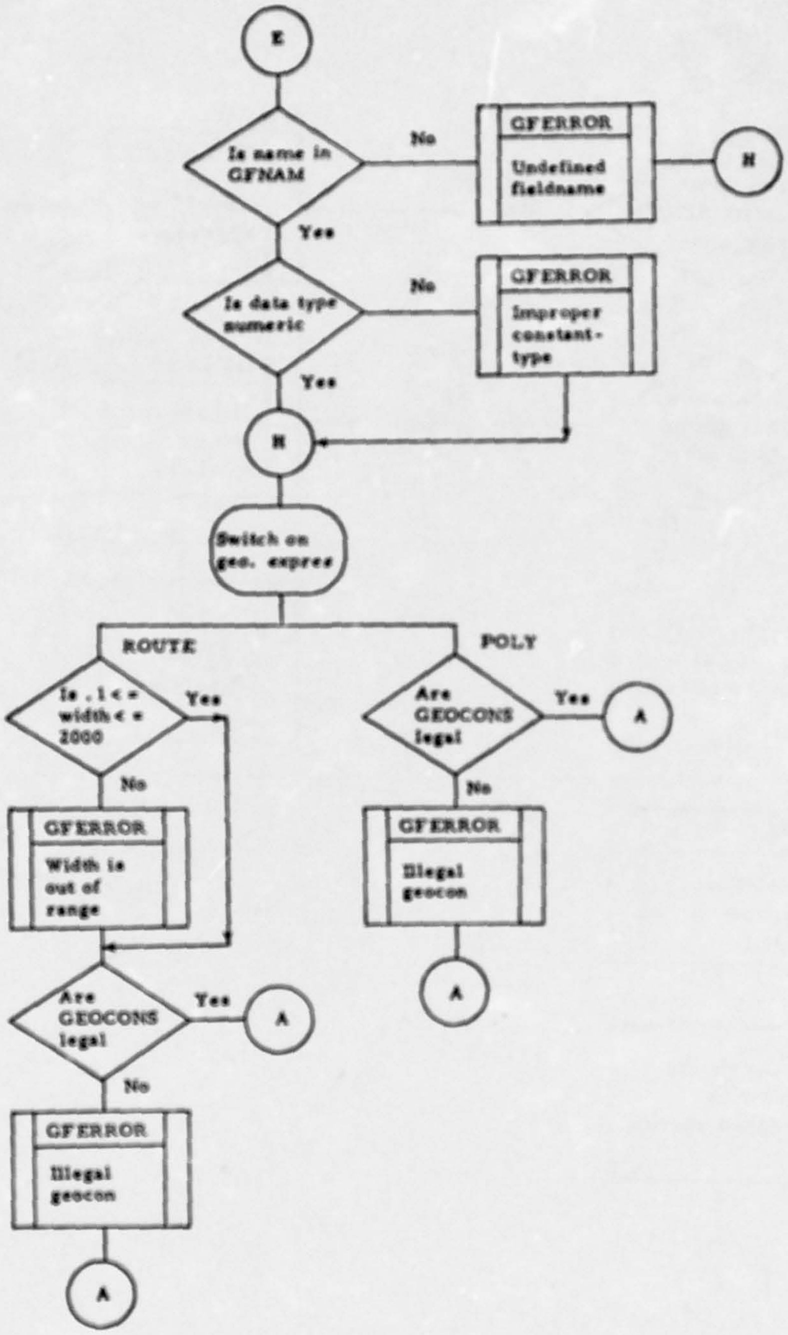


Figure 36. VALIDATE Process Data Flow (Sheet 15 of 16).

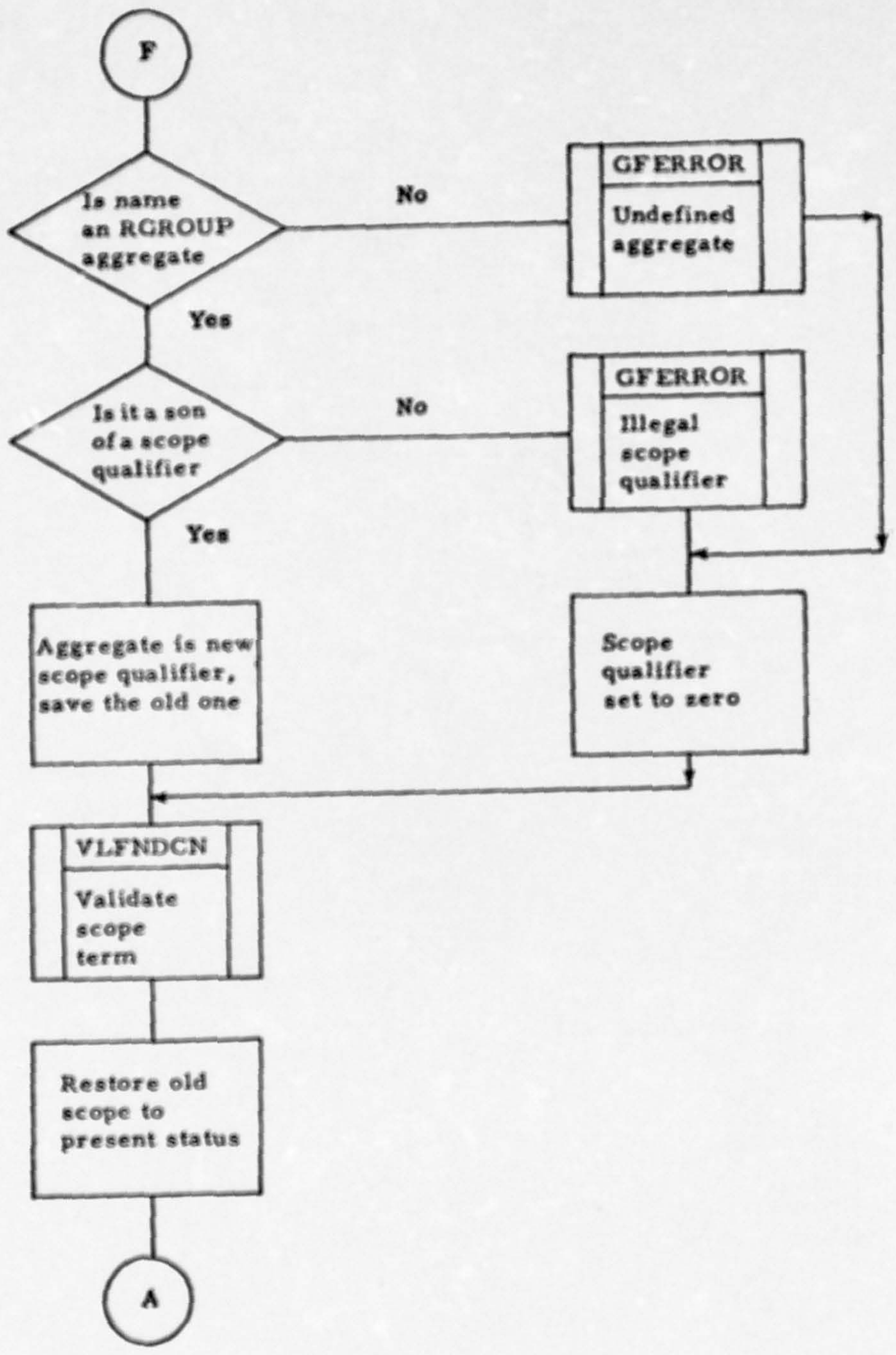

F
Is name an RGROUP aggregate
No
Yes
GFERROR
Undefined aggregate
Is it a son of a scope qualifier
No
Yes
GFERROR
Illegal scope qualifier
Aggregate is new scope qualifier, save the old one
Scope qualifier set to zero
VLFNDCN
Validate scope term
Restore old scope to present status
A

Figure 36. VALIDATE Process Data Flow (Sheet 16 of 16).

GTREE –	parse tree. Used by all statements.
GTDAT –	parse tree data table. Used by all statements except in the special cases involving CLOSE and VIEW.
GFNAM –	global field name table. Used by statements DISPLAY and FIND.
GANAM –	global aggregate name table. Used by statements DISPLAY and FIND.
GFDES –	global field description table. Used by statements DISPLAY and FIND.
GADES –	global aggregate description table. Used by statements DISPLAY and FIND.

GENERAL PROCESS FLOW – VALIDATE performs semantical validation upon the parse tree (GTREE) which is built by EXEC. Control is passed to VALIDATE when EXEC determines that a syntactically correct input statement requires validation. Four arguments are passed to VALIDATE, all of which are file descriptors for the following global data files: GTREE, GTDAT, GLABL, and GLIST.

VALIDATE reads in the parse tree (GTREE) and picks up the first token type. This token type defines the UDL statement type and subsequently the semantic rules against which VALIDATE tests the statement. VALIDATE continues to work down the parse tree, validating the statement in accordance with a prespecified set of validation rules. When encountering a valid terminal token type, the parse tree may be modified to more closely define the token. For example, a DISPLIST NAME in a DISPLAY statement is validated as either an aggregate name or a field name. If the name is indeed legal, the appropriate token type of either AGGNAME or FLDNAME is exchanged for NAME in GTREE. Whenever possible, the name's index in its associated global table is inserted after the newly assigned token type.

For the foregoing example, GTREE may be altered as follows:

Before Validation		After Validation
NAME	GTREE [n]	AGGNAME
Pointer to the first letter of NAME in GTDAT	GTREE [n + 1]	Index in both GANAM and GADES
Number of characters	GTREE [n + 2]	Not applicable

These indexes expedite further processing of the statement in the parse tree by allowing direct indexing into global files.

VALIDATE attempts to catch all errors and to print meaningful diagnostic messages. The error messages are printed by GFERROR, the global error routine which accepts error codes as input. An optional second input is a pointer to a character string which is inserted into the error message.

On completion of VALIDATE, the process returns an integer value to EXEC which indicates the status of the statement. If the statement contains no errors, EXEC passes control to the appropriate module for final processing. If, however, a user error status exists, the statement is abandoned and EXEC waits for further interaction with the user.

The remaining discussion of VALIDATE is broken down by individual statement types. The semantic rules, as well as program details, are discussed for each statement type. The seven statement types are OPEN, SAVE, CLOSE, DELETE, VIEW, DISPLAY, and FIND.

a. OPEN – The token OPEN points to one leg, OPENLIST, which in turn points to many OPENITEMs. Each OPENITEM has two NAMEs: the first confirmed as an existing FILENAME, and the second as an appropriate ACCESS code.

b. SAVE – The token SAVE points to two NAMEs. The first NAME is validated to be an existing TAG; the second NAME is validated to be neither an existing label nor listname.

c. CLOSE – The token CLOSE may have one or no legs. Validation is finished in the latter case. In the former case, the leg points to token type FNAMLIST which in turn points to one or more NAMEs. Each NAME is validated to be an existing FILENAME.

d. DELETE – The DELETE token points to a token type which is either SCHEMA or LIST. The leg of this token points to a NAME which is validated as an existing FILENAME or LISTNAME, respectively.

e. VIEW – the VIEW token points to a token type which is either SCHEMA, FILE, or LIST. The leg of the SCHEMA token points to a NAME which is validated as an existing FILENAME.

f. DISPLAY – The first leg of the DISPLAY token points to NAME, confirmed to be either an existing TAG or LISTNAME. The second leg points to a token type of either REMOTE (nothing further has to be validated) or DISPLIST.

DISPLIST has any number of legs, each pointing to a token type of CHARCONS, TREE, NAME, or SNAME.

CHARCONS (character constant string) requires no validation.

TREE has one leg pointing to a NAME which is confirmed to be an AGGNAME.

NAME is validated to be either an AGGNAME or a FLDNAME.

SNAME points to a NAME which is checked to be a subscripted field whose number of subscripts is the number of levels of arrays in which the field is located. Each subscript must be within the range defined for the associated array.

The display attribute of all fields must be visible.

g. FIND – the label on the FIND statement is validated to be a name different from those existing user tag and listnames.

The first leg of the FIND statement must specify either a filename or a statement label of another FIND statement; the token is either IN or SOURCE. The name which IN points to is required to be an existing filename, and the statement is completely "independent" of previous FIND statements. SOURCE, on the other hand, points to a name which is validated to be an existing tag (GLABL). The statement itself is "dependent" upon a previous inquiry labelled by the aforementioned tag. A dependent FIND statement can not reference another dependent FIND statement.

The selection-criteria of the FIND statement is nothing more than a specialized boolean expression for the explicit purpose of record selection. The legal UDL boolean operators are OR, AND, XOR, NOR, and NOT. No validation is done at this level although, at the end of the validation of the statement, boolean expressions are normalized and simplified by the boolean expression normalizer (BEN). Refer to appendix C for details.

The argument of a boolean operator is either a selection term or another selection-criteria enclosed in parentheses with an optional scope-qualifier.

The scope-qualifier must be the name of a repeating group and the fields referenced in the qualified selection-criteria must be directly subordinate to the scope-qualifier. Scope-qualifiers can also be nested to force the desired lineage path during the selection process. Again, those nested scope-qualifiers must be directly subordinate repeating-group names.

The three types of relational terms are:

1. Standard relational terms.
2. Range relational terms.
3. Geographic relational terms.

The five standard relational operators LT, GT, LE, GE, and EQ are numeric relational operators. EQ can also be used as a character relational operator.

The range relational operators are WRG and ORG. The WRG operator requires the field to be within or equal to the limits set by the two given numeric constants. Similarly, the ORG operator requires the field to be outside the two numeric constants. The first expression specifies the lower range limit; the second, the upper limit.

The last relational operator, the geographic operator, applies to the relationships of fields to circles, routes, or polygons on the surface of the earth.

The INSIDE operator requires the field's geographic value to be inside the area specified by a circle or a polygon. Similarly, the OUTSIDE operator demands that the field's values be outside the specified area of a polygon. Finally, the ALONG operator requires the field's value to be on the perimeter of either a route or polygon. CIRCLE is defined by a radius and a center, and POLY is designated by a series of geographic constants. ROUTE is established by the series of geographic constants on which it lies and the bandwidth of the route.

Refer to table 1 for more specific details on the validating of FIND statements.

TABLE I. SELECTION-CRITERIA VALIDATION.

Keyword	First Leg	Second Leg	Third Leg	Explanation of Validation
SCOPE	NAME	–	–	Aggregate name of a repeating group.
EQ	NAME* or SNAME	NUMCONS or INTEGER or	–	Field's data type is numeric.
		CHARCONS	–	Field's data type is character. Unless field type is single-value variable, the number of characters in CHARCONS is less than or equal to GSIZE.
LT, LE, GT, GE	NAME* or SNAME	NUMCONS or INTEGER	–	Field's data type is numeric.
WRG, ORG	NAME* or SNAME	NUMCONS or INTEGER	NUMCONS or INTEGER	Field's data type is numeric, first number $\leq$ second number.
INSIDE	NAME* or SNAME	CIRCLE or POLY	–	Field's data type is geographic.
OUTSIDE	NAME* or SNAME	POLY	–	Field's data type is geographic.
ALONG	NAME* or SNAME	ROUTE or POLY	–	Field's data type is geographic.
CIRCLE	NUMCONS or INTEGER (radius)	GEOCONS	–	$0.1 \leq$ radius ≤ 2000.
POLY	Pointer to a series of GEOCONS	–	–	$3 \leq$ number of GEOCONS ≤ 506.

TABLE I. SELECTION-CRITERIA VALIDATION. (Cont)

Keyword	First Leg	Second Leg	Third Leg	Explanation of Validatio
ROUTE	NUMCONS or INTEGER (bandwidth)	Pointer to a series of GEOCONS	–	$0.1 \leq$ bandwidth ≤ 2000. $3 \leq$ number of GEOCONS ≤ 506.
NAME*	–	–	–	Field name whose parent is not an array.
SNAME	–	–	–	Field name. The number of subscripts is the correct levels of subscript for the field. Subscript number is within the occurrence size (GOCCS).
For both NAME* and SNAME	–	–	–	Keyed field if tag is independent. Keyed or dependent if tag is dependent. If there is a scope-qualifier, it is the parent of the field.
GEOCONS	–	–	–	$0 \leq$ degrees for latitude ≤ 90. $0 \leq$ degrees for longitude ≤ 180. $0 \leq$ minutes ≤ 59. $0 \leq$ seconds ≤ 59.

COMMAND (Miscellaneous Commands Processor)

The COMMAND processor processes the UDL DELETE and LOGOFF commands. It modifies the necessary global data to reflect the specified command.

GLOBAL DATA USAGE – COMMAND uses the following global data:

GTREE – parse tree.
GUSER – user descriptions (LOGOFF).
GFILD – file descriptions (DELETE SCHEMA).
GFILN – file names (DELETE SCHEMA).
GLIST – list names and descriptions (DELETE LIST).

LOCAL DATA USAGE – COMMAND has no pertinent local data.

GENERAL PROCESS FLOW – COMMAND is activated whenever a user has input a DELETE or LOGOFF command. COMMAND first reads the parse tree (GTREE) in order to determine the type of command. If the command is LOGOFF, COMMAND locks the user descriptions (GUSER) via GFLOCK and reads GUSER. The user's status is set to logged-off, and COMMAND then writes GUSER and unlocks GUSER via GFUNLOCK. If the command is DELETE LIST, COMMAND reads the user's list descriptions (GLIST). The UNIX file names for the specified list's record map and record data (GRMAP, GRDAT) are determined and those files deleted. COMMAND then deletes the specified list's name and description and writes GLIST back out. If the command is DELETE SCHEMA, COMMAND validates that the user is the "superuser." If not, a diagnostic is output and COMMAND terminates. Otherwise, COMMAND reads the file names and descriptions (GFILN, GFILD). The UNIX file names for the specified target system file's aggregate names and descriptions (GANAM, GADES) and field names and descriptions (GFNAM, GFDES) are determined and those files deleted. COMMAND then deletes the specified target system file's name and description and writes GFILN and GFILD back out. Figure 37 supplies more information.

VIEW

The VIEW process processes the UDL VIEW statement. It displays either list names and their associated file names, file names and their associated data base names, a specific file's structure, or batch transaction information or data according to the VIEW argument.

GLOBAL DATA USAGE – VIEW uses the following global data:

GTREE – parse tree.
GFILN – file names.
GFILD – file descriptions (VIEW SCHEMA and VIEW FILE).
GDBASE – data base names (VIEW SCHEMA and VIEW FILE).
GLIST – list names and descriptions (VIEW LIST).
GFNAM – field names (VIEW SCHEMA).
GFDES – field descriptions (VIEW SCHEMA).
GANAM – aggregate names (VIEW SCHEMA).
GADES – aggregate descriptions (VIEW SCHEMA).
GUSER – user descriptions (VIEW TRAN).
GTDES – transaction descriptions (VIEW TRAN).

LOCAL DATA USAGE – VIEW uses VILINE – character array in which each print line is constructed.

GENERAL PROCESS FLOW – VIEW is activated whenever a user has input a VIEW command. VIEW first reads the parse tree (GTREE) and the file names (GFILN). If the VIEW argument is FILE or SCHEMA, the file descriptions (GFILD) are read in. VIEW then switches on the VIEW

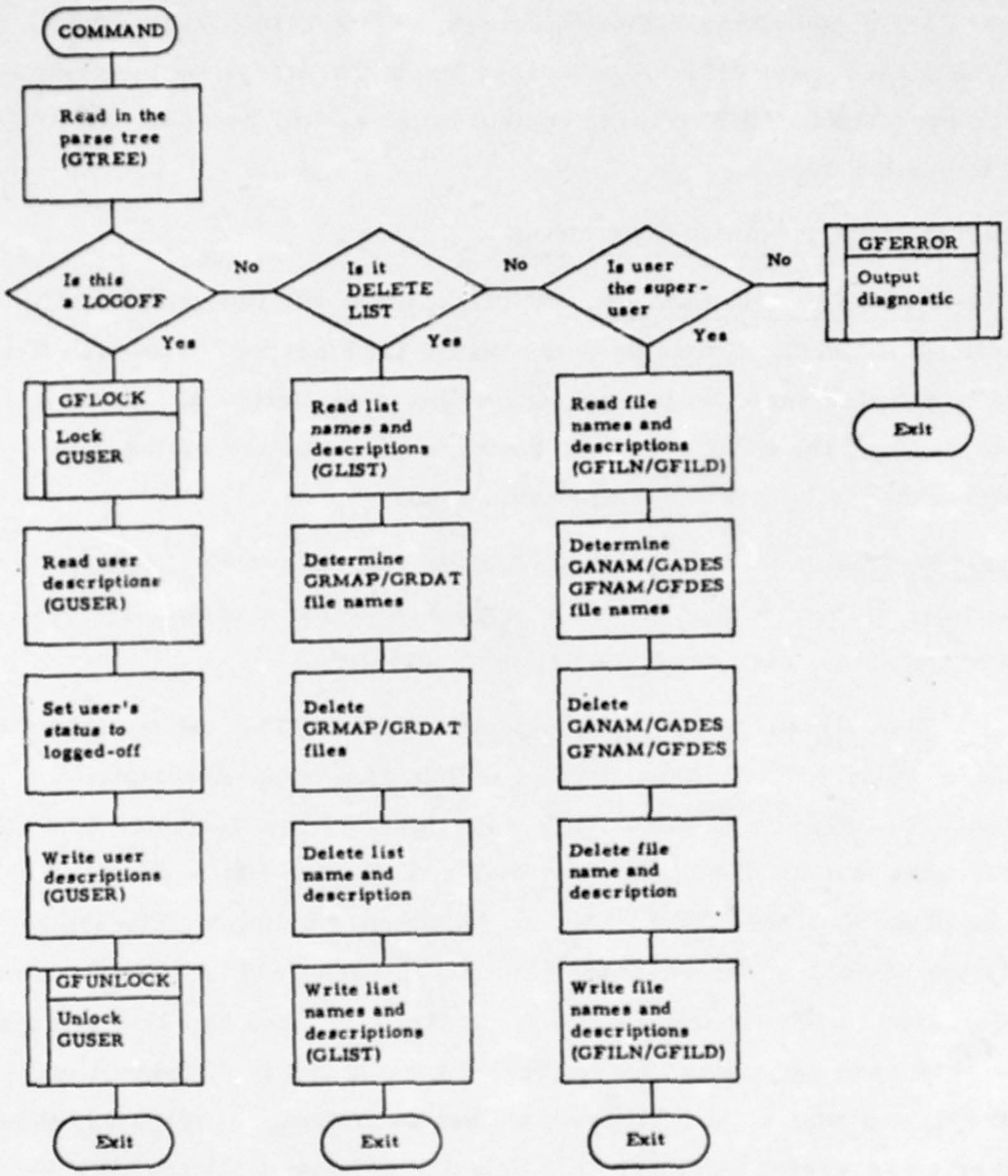

COMMAND
Read in the parse tree (GTREE)
Is this a LOGOFF
No
Yes
Is it DELETE LIST
No
Yes
Is user the super-user
No
Yes
GFERROR
Output diagnostic
Exit
GFLOCK
Lock GUSER
Read user descriptions (GUSER)
Set user's status to logged-off
Write user descriptions (GUSER)
GFUNLOCK
Unlock GUSER
Exit
Read list names and descriptions (GLIST)
Determine GRMAP/GRDAT file names
Delete GRMAP/GRDAT files
Delete list name and description
Write list names and descriptions (GLIST)
Exit
Read file names and descriptions (GFILN/GFILD)
Determine GANAM/GADES GFNAM/GFDES file names
Delete GANAM/GADES GFNAM/GFDES
Delete file name and description
Write file names and descriptions (GFILN/GFILD)
Exit

Figure 37. COMMAND Process Data Flow.

argument and the VIEW command is processed as follows: VISCHEMA processes VIEW SCHEMA, VIFILES processes VIEW FILE, VILIST processes VIEW LIST, and VITRAN processes VIEW TRAN. After the command has been processed, VIEW returns control to the calling process. See figure 38 for data flow.

MAJOR FUNCTION DESCRIPTIONS –

VILIST (Display List Names) – VILIST reads in the list names and descriptions (GLIST). It then outputs a VIEW LIST header. For each list in GLIST, a test is made as to whether the list is explicit; i.e., user-defined. If it is, the list's name and the name of the target system file from which the list's data came are both output.

VIFILES (Display File Names) – VIFILES outputs a VIEW FILE header. For each file in the system, the file's name and the name of the data base (target system) to which it belongs are both output.

VISCHEMA (Display a File's Structure) – VISCHEMA first reads in the data needed to describe a file's structure: field names and descriptions (GFNAM, GFDES) and aggregate names and descriptions (GANAM, GADES). It then outputs the VIEW SCHEMA header containing the file's name and the name of the data base (target system) to which it belongs. The field data for the basic data set are then output. For each field, the field's name, type, data type, interrogation attribute, display attribute, and size are displayed. For each aggregate, the aggregate's name and occurrence count (for arrays) are output. If any aggregate has subordinate aggregates, those subordinate aggregates and their field data are displayed. After all subordinate aggregates are displayed, any sibling aggregates and their subordinate aggregates are displayed. The file structure is presented in a hierarchical format.

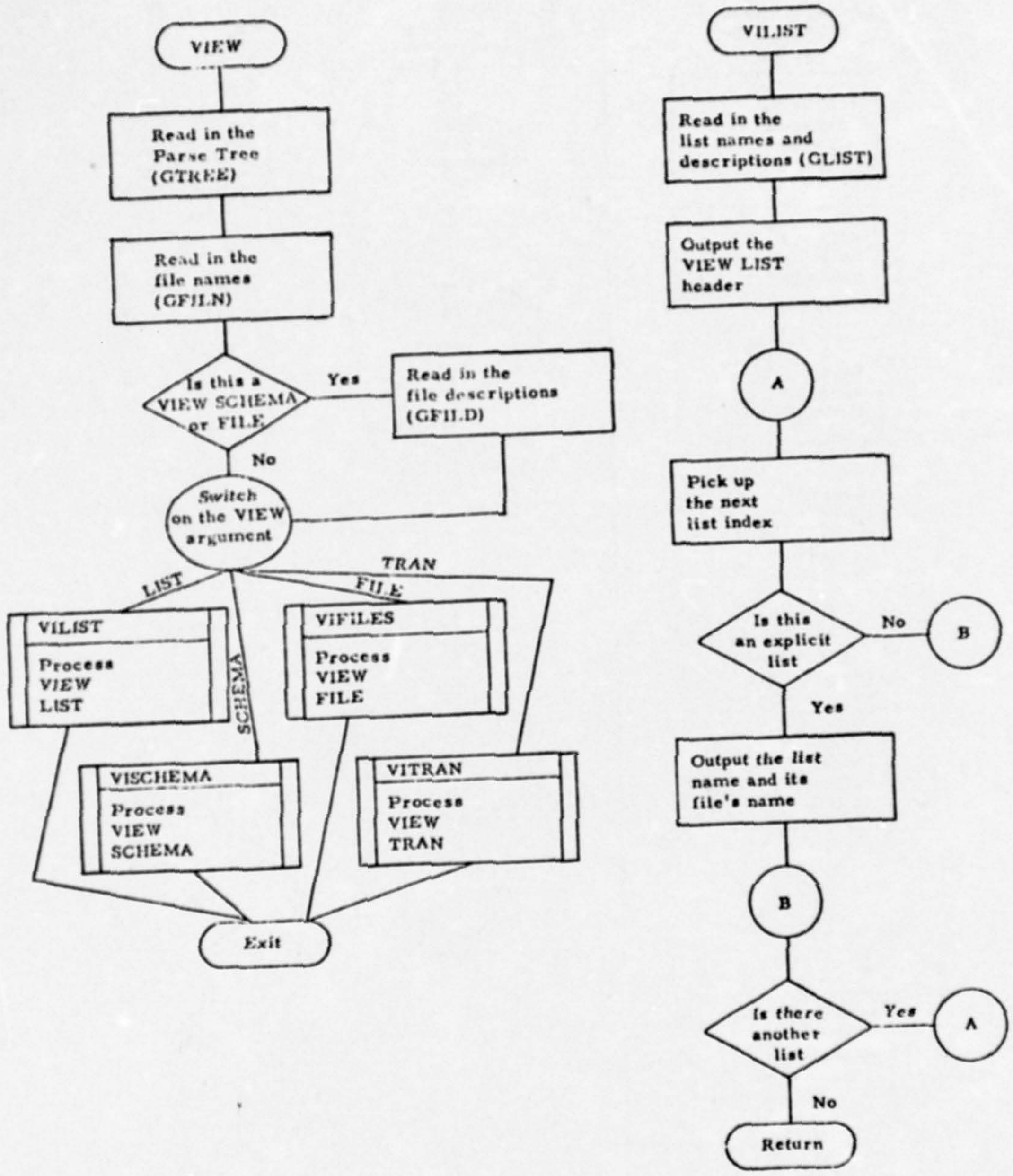

VIEW
Read in the Parse Tree (GTREE)
Read in the file names (GFILN)
Is this a VIEW SCHEMA or FILE
Yes
Read in the file descriptions (GFILD)
No
Switch on the VIEW argument
LIST
SCHEMA
FILE
TRAN
VILIST Process VIEW LIST
VIFILES Process VIEW FILE
VISCHEMA Process VIEW SCHEMA
VITRAN Process VIEW TRAN
Exit
VILIST
Read in the list names and descriptions (GLIST)
Output the VIEW LIST header
A
Pick up the next list index
Is this an explicit list
No
B
Yes
Output the list name and its file's name
B
Is there another list
Yes
A
No
Return

Figure 38. VIEW Process Data Flow (Sheet 1 of 4).

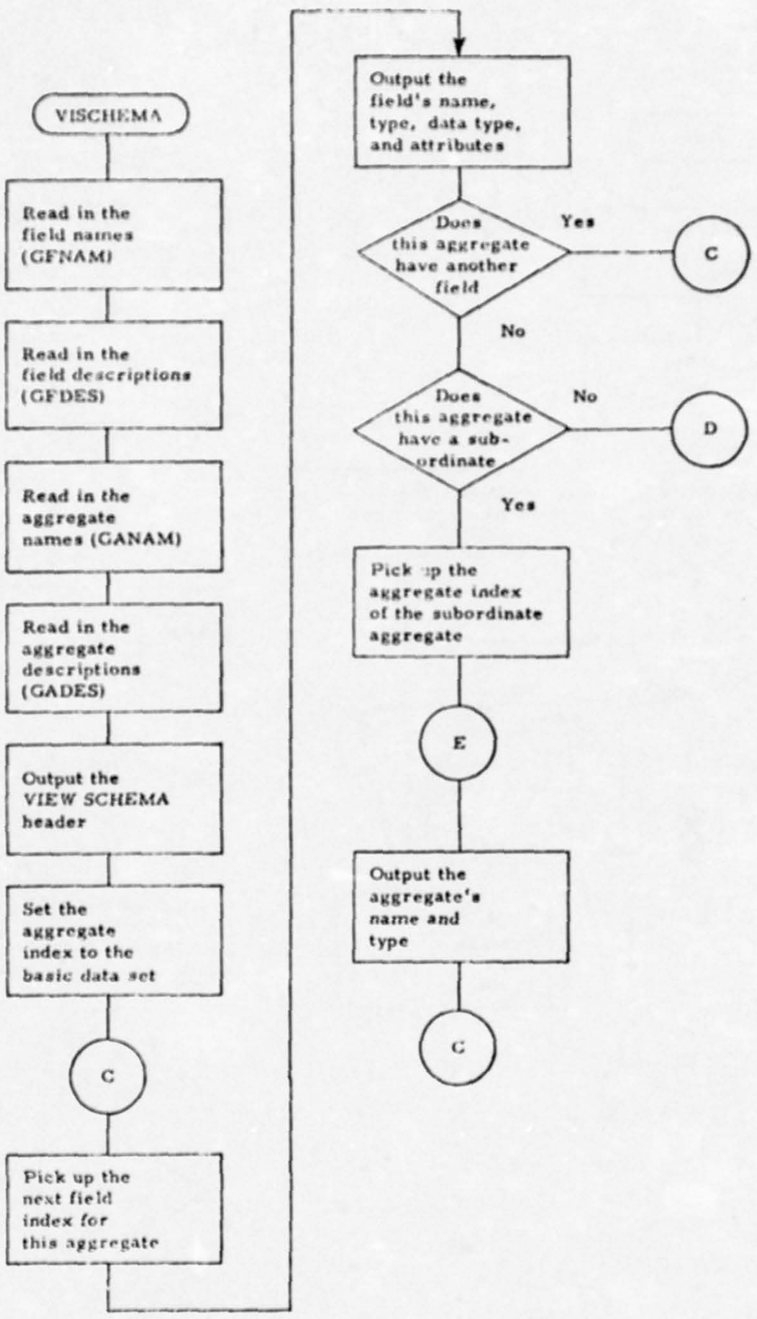

VISCHEMA
Read in the field names (GFNAM)
Read in the field descriptions (GFDES)
Read in the aggregate names (GANAM)
Read in the aggregate descriptions (GADES)
Output the VIEW SCHEMA header
Set the aggregate index to the basic data set
C
Pick up the next field index for this aggregate
Output the field's name, type, data type, and attributes
Does this aggregate have another field
Yes
C
No
Does this aggregate have a subordinate
No
D
Yes
Pick up the aggregate index of the subordinate aggregate
E
Output the aggregate's name and type
C

Figure 38. VIEW Process Data Flow (Sheet 2 of 4)

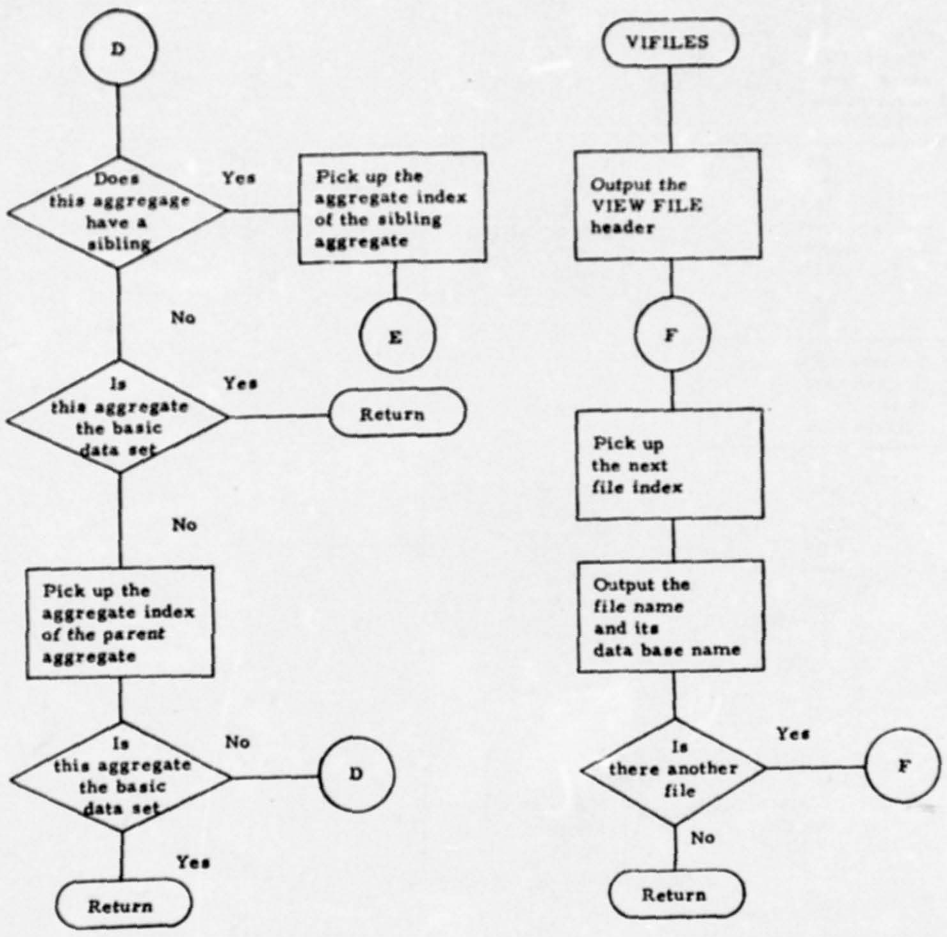

D
Does this aggregage have a sibling
Yes
Pick up the aggregate index of the sibling aggregate
E
No
Is this aggregate the basic data set
Yes
Return
No
Pick up the aggregate index of the parent aggregate
Is this aggregate the basic data set
No
D
Yes
Return
VIFILES
Output the VIEW FILE header
F
Pick up the next file index
Output the file name and its data base name
Is there another file
Yes
F
No
Return

Figure 38. VIEW Process Data Flow (Sheet 3 of 4).

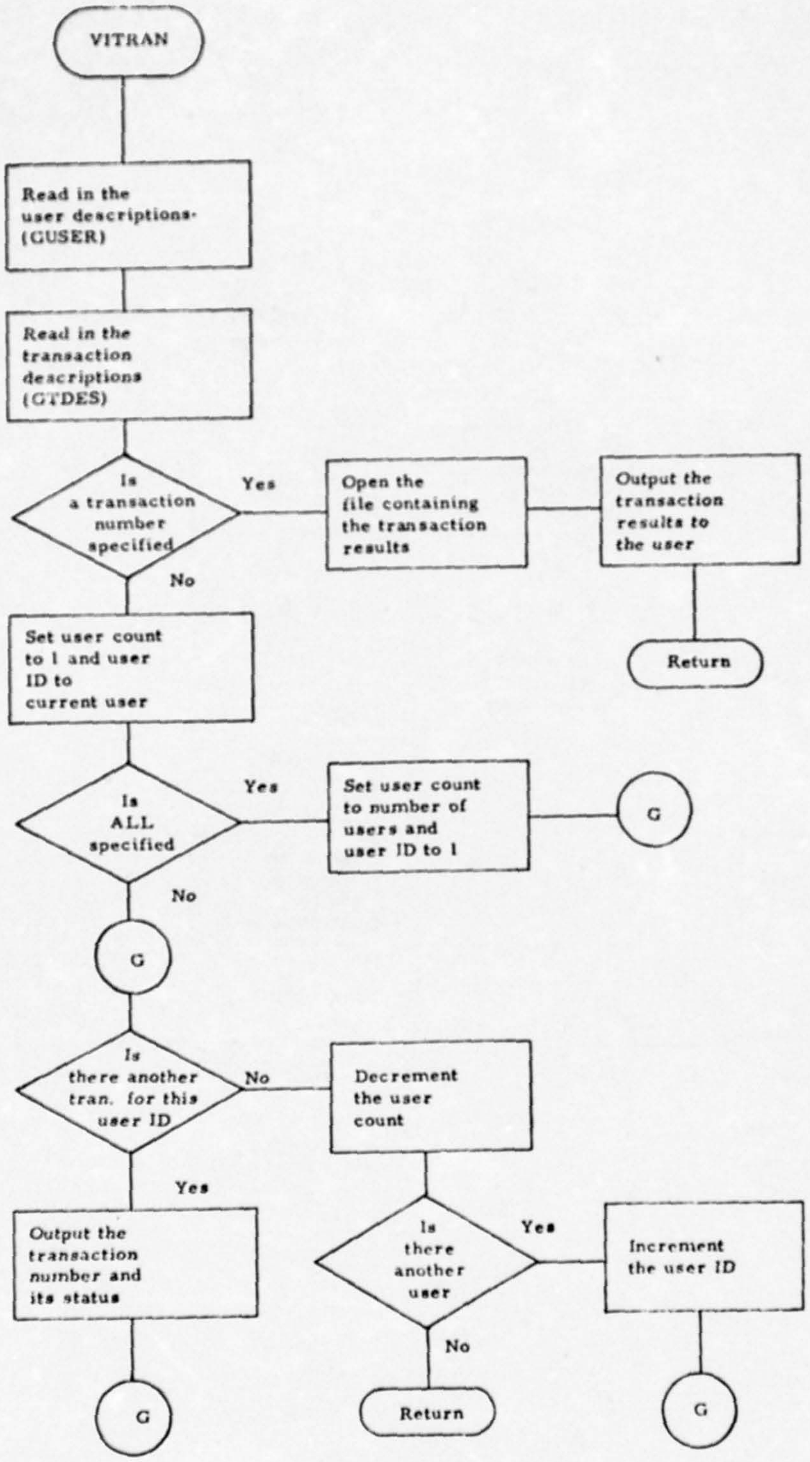

VITRAN
Read in the user descriptions (GUSER)
Read in the transaction descriptions (GTDES)
Is a transaction number specified
Yes
No
Open the file containing the transaction results
Output the transaction results to the user
Return
Set user count to 1 and user ID to current user
Is ALL specified
Yes
No
Set user count to number of users and user ID to 1
G
G
Is there another tran. for this user ID
No
Yes
Decrement the user count
Output the transaction number and its status
Is there another user
Yes
No
Increment the user ID
G
Return
G

Figure 38. VIEW Process Data Flow (Sheet 4 of 4).

VITRAN (Display Batch Transaction Data) – VITRAN reads the user descriptions (GUSER) and the transaction descriptions (GTDES). If a specific transaction is requested by a user and that transaction is complete, the applicable transaction results file is opened and the contents displayed to the user. If a user is requesting the status of all his transactions, each transaction number with its status is displayed. If the superuser is requesting the status of all transactions, each user name is displayed with his transaction numbers and their statuses.

DDL (Data Definition Language Process)

The DDL process generates UDL file dictionary definitions for a given file as defined by a set of DDL statements input by a user. The Data Definition Language (DDL) provides the user with a facility for defining a UDL file's composition for subsequent use with the Uniform Data Language (UDL). DDL is responsible for the lexical and syntactic analysis, as well as actual statement processing, of the DDL statement set.

GLOBAL DATA USAGE – DDL uses the following global data tables:

GANAM – aggregate names (created by DDL).

GADES – aggregate descriptions (created by DDL).

GFNAM – field names (created by DDL).

GFDES – field descriptions (created by DDL).

GFILN – file names.

GFILD – file descriptions.

GTREE – parse tree.

GTDAT – parse tree data.

GDBASE – data base identification (in SCHEMA statement).

LOCAL DATA USAGE – The following local data are used by DDL:

DDSTACK – Structure DDSTACK follows the family lineages of aggregates to calculate relative indexes which point into the internal record map (GRMAP) or into the internal record data array (GRDAT) for aggregates and fields.

LXKYWRD – KEYWORDS.

LXTOKEN – KEYWORD token types.

GENERAL PROCESS FLOW – EXEC calls DDL when it encounters either a SCHEMA statement or an EXECUTE SCHEMA statement. The EXECUTE SCHEMA statement provides users with the facility to generate UDL file definitions in an ordinary UNIX file, which can then be read by DDL for syntactic and semantic validation only, or for actual file dictionary generation. On the other hand, the encountering of a SCHEMA statement by EXEC implies an interactive environment with a user's terminal. If a UNIX file is being used, DDL calls the LEX/parser (LXINIT) to process the first line (a SCHEMA statement) in the file. Henceforth, DDL handles both modes of input identically.

The SCHEMA statement must appear once in the file definition, and it must be the first line of the definition.

Pertinent information, such as the filename and the data base identifier, is stored in the SCHEMA statement. The filename is authenticated to be a unique filename and the data base identifier indicates the target system where the file actually resides. The identifier is verified to be the name of a target system to which ADAPT I maps.

At the conclusion of processing any DDL statement, the LEX/parser is called upon to syntactically validate the next statement and to build the parse tree (GTREE) and the parse tree data (GTDAT).

It might be noted that throughout the DDL process an error message is printed whenever a syntactic or semantic user error is encountered. DDL attempts to catch all errors in the schema-definition-block, as well as to find the first error. The error messages are output by the global function, GFERROR.

Following the SCHEMA statement is the schema-clause, a block of data structure definitions making up the file's logical structure. Within the block are two basic types of definitions: those defining fields and those defining aggregates. Aggregate-definitions are optional depending on whether a file has them, but a field-definition-block is mandatory. In other words, every file has a basic data set which contains at least one field. The basic data set definition is a sequence of field-definition statements and must occur first in the schema-clause.

A field-definition statement is indicated by the token type FIELD, followed by the field's name and a field-clause. This fieldname must be a unique name within the file. The mandatory field-clause specifies the field's type, its data-type, its access attributes, and its size. Two types of fields exist in UDL: single-valued (SV) and multivalued (MV). Following the field-type is the field's data-type specification. One of the following three data-types must be specified: character (CHAR), numeric (NUM), or geographic (GEO). Each field also has associated with it an interrogation attribute and a display attribute. Valid interrogation attributes are KEY, for fields which can be referenced in an initial FIND statement; NKEY (non-key), for fields which can not be referenced in FIND statements; and DEP (dependent) for fields which can only be referenced in dependent FIND statements. Two display attributes are recognized in UDL: visible (VIS) and invisible (INV). A field can not logically be nonkeyed and dependent simultaneously.

The maximum field size (as an ASCII representation, regardless of the data-type) must be specified for each field in the form of an integer or a V (which symbolizes a variable length character string). A variable length character string is legal if the field is a visible, nonkeyed, and single-valued field whose data-type is character. If an integer is given for the field size and if the field is single valued, the field's GIPTR is set to the parent aggregate GSIZE. The parent aggregate GSIZE is increased by this field size.

Aggregates are composed of either arrays or repeating-groups. When present in the file, they must be specified following the basic data set definition block. Subroutine DDAGG processes both types of aggregate-definition-blocks, distinguishing only when necessary. An array-definition-block begins with the token ARRAY; similarly, a repeating-group-definition-block begins with the token RGROUP. If a previous aggregate-definition existed, a check is made to ensure the proper closure of it with either an EARRAY or an ERGROUP. The aggregate statement is comprised of the aggregate name, its order (for arrays only), and an optional parent aggregate name. The aggregate name is confirmed to be either an unique name within the file or an undefined aggregate. At this point the array-statement specifies its order, which is defined to be the maximum number of occurrences allowed for the array. The final specification for both statement types is the name of another aggregate, the parent name. (The parent name is not necessarily defined at this point, thus relieving the user of defining aggregates in the restrictive parent-subordinate order.) The necessary family links (parent, subordinate, and sibling) are made to give shape to a hierarchical structure. If no parent-name is specified, the parent is the basic data set.

The array-definition-block must be terminated by the token EARRAY; the repeating-group-definition by ERGROUP. Upon the recognition of these tokens, two checks must be made. The first check is to validate that an aggregate definition does indeed exist. If it does, verification is made that at least one field is defined in the aggregate-definition-block.

The final statement in a schema-definition is ESCHEMA. Upon recognition of the ESCHEMA token, the DDL module makes a final check and, finding no preceding errors, writes out the pertinent files.

The final pass checks for the following errors:

a. Improper closure of the last aggregate-definition-block.

b. Undefined aggregates.

c. An aggregate which directly or indirectly (through other aggregates) is its own parent.

d. Repeating-groups whose parents are arrays (arrays can not be parents of repeating-groups).

The relative indexes which point into the internal record map (GRMAP) or into the internal record data array (GRDAT) are calculated and stored. The pushdown-popup stack (DDSTACK) is used to follow the lineages of aggregates, starting with the basic data set. An aggregate's subordinates are assigned their GMPTRs in the consecutive order of their input (as indicated by sibling links). Having assigned the subordinates their pointers, the aggregate's multivalued and single-valued variable length fields are now assigned their GIPTRs.

A schema-definition is not a valid definition if any errors were detected throughout the processing of the schema-definition. If any errors were noted or the mode is validation only, control is returned to EXEC. If, however, the schema-definition is error free, DDL writes out the file's dictionary definitions. These files include the information concerning the aggregates (GANAM and GADES) and the fields (GFNAM and GFDES). Finally, DDL stores the filename and description in the file name and file description (GFILN and GFILD) tables. DDL returns control to EXEC. See figure 39 for more information.

MAJOR FUNCTION DESCRIPTIONS –

LXINIT (LEX/Parse Initialization) – LXINIT is merely an initialization function for the LEX/parse portion of this process. It initializes pertinent data including a flag which it returns to the calling function to indicate either end-of-file (EOF), syntax error, or statement accepted. LXINIT calls the parsing function (YYPARSE) and returns with the flag value.

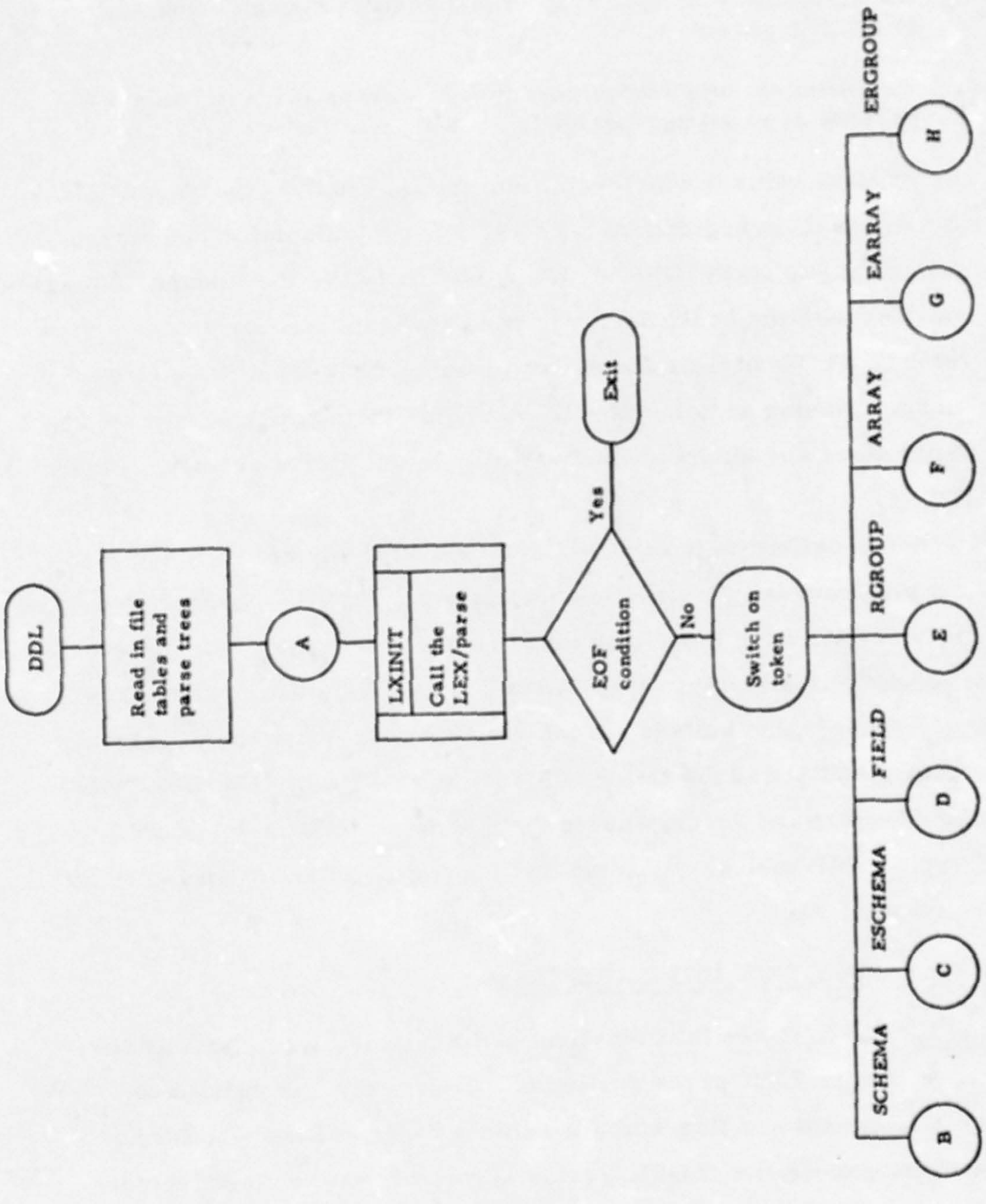


Figure 39. DDL Process Data Flow (Sheet 1 of 9).

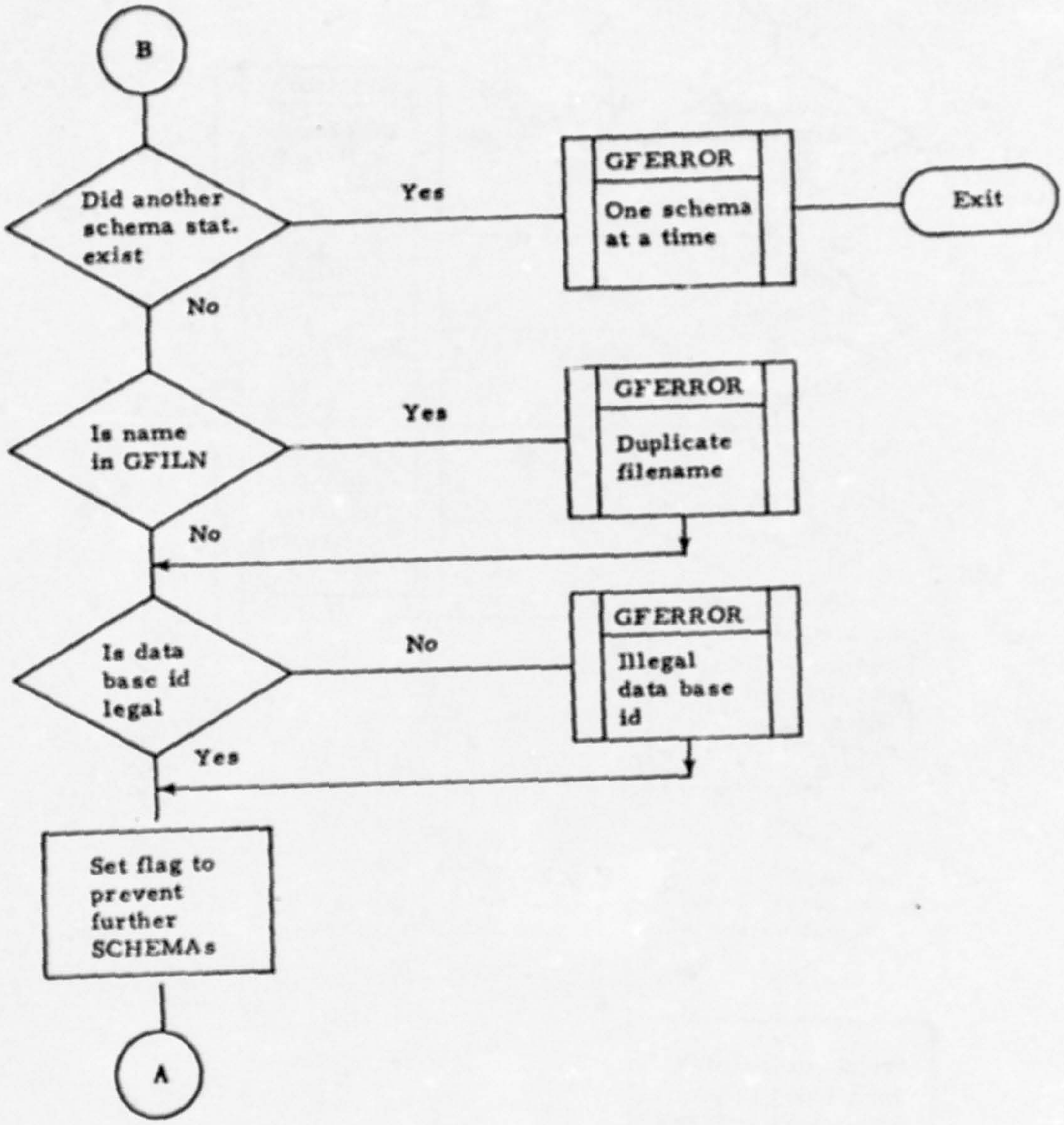


Figure 39. DDL Process Data Flow (Sheet 2 of 9).

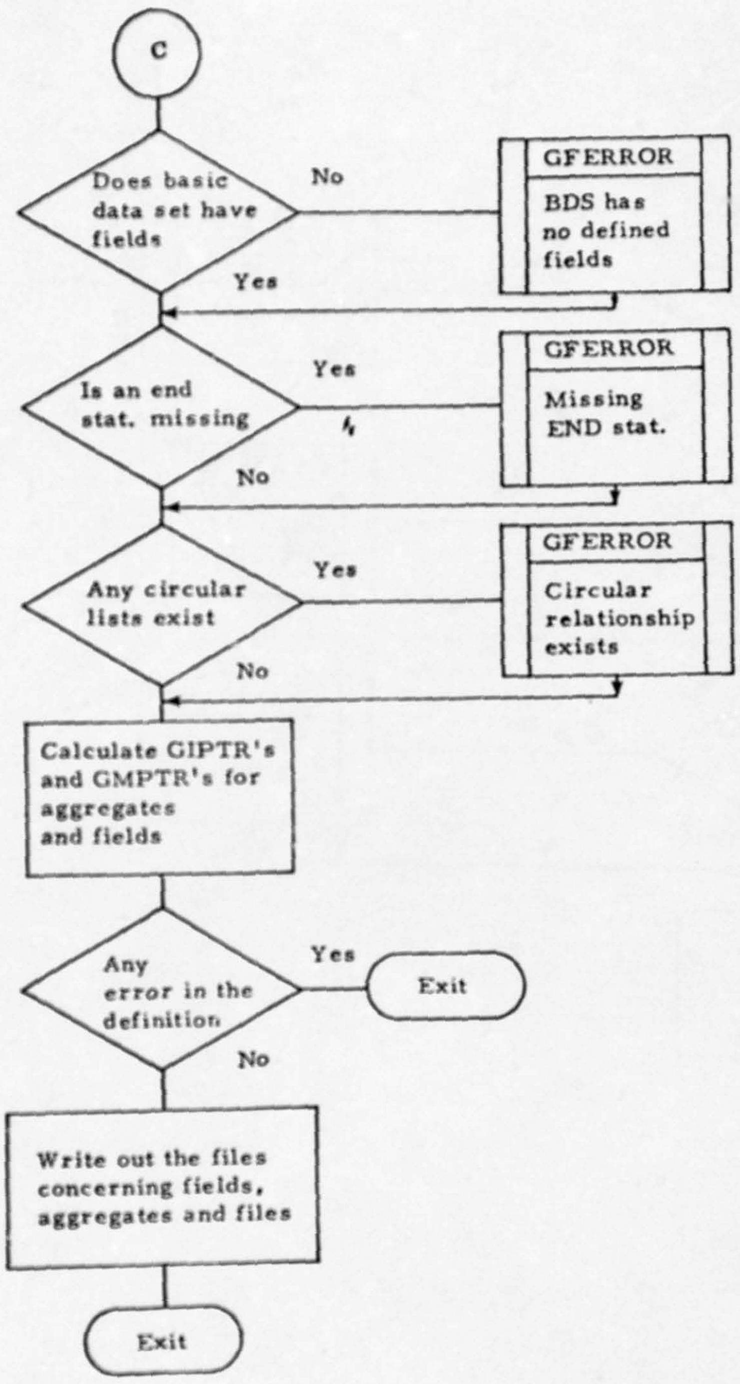

C
Does basic data set have fields
No
Yes
GFERROR
BDS has no defined fields
Is an end stat. missing
Yes
No
GFERROR
Missing END stat.
Any circular lists exist
Yes
No
GFERROR
Circular relationship exists
Calculate GIPTR's and GMPTR's for aggregates and fields
Any error in the definition
Yes
No
Exit
Write out the files concerning fields, aggregates and files
Exit

Figure 39. DDL Process Data Flow (Sheet 3 of 9).

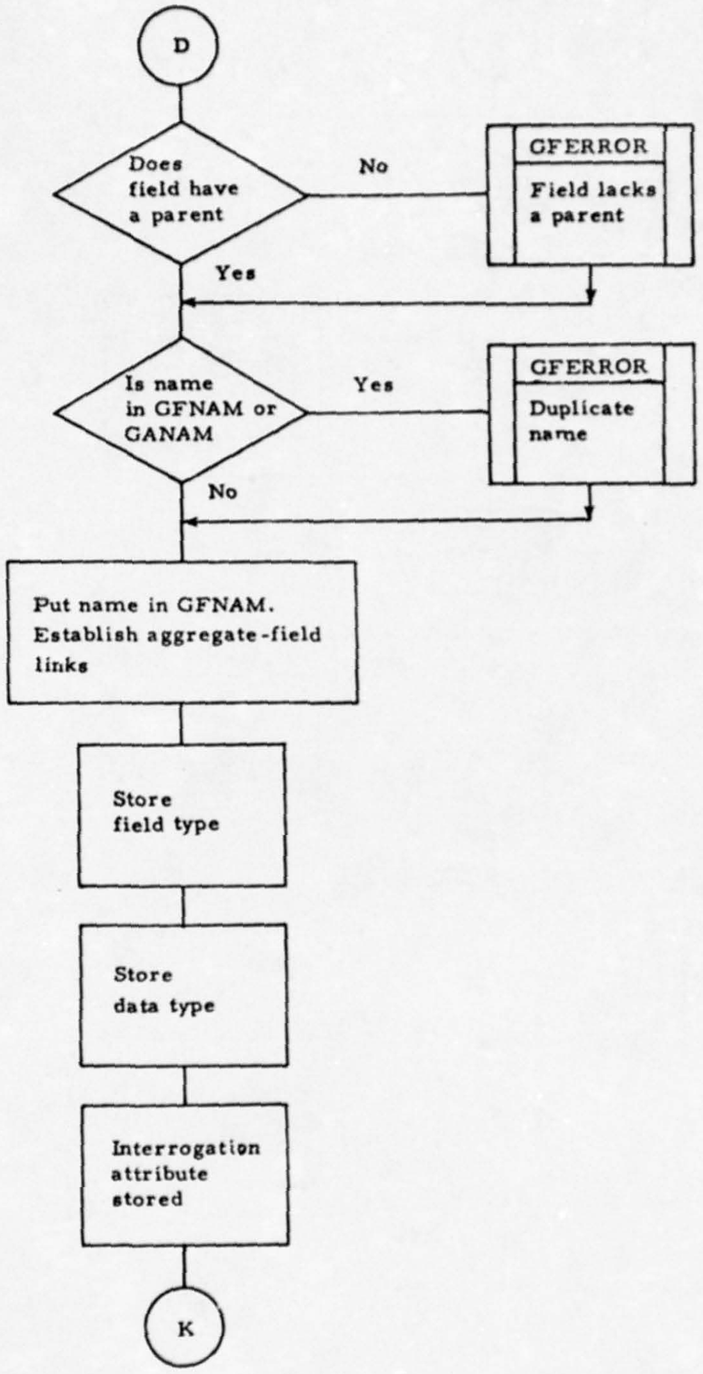


Figure 39. DDL Process Data Flow (Sheet 4 of 9).

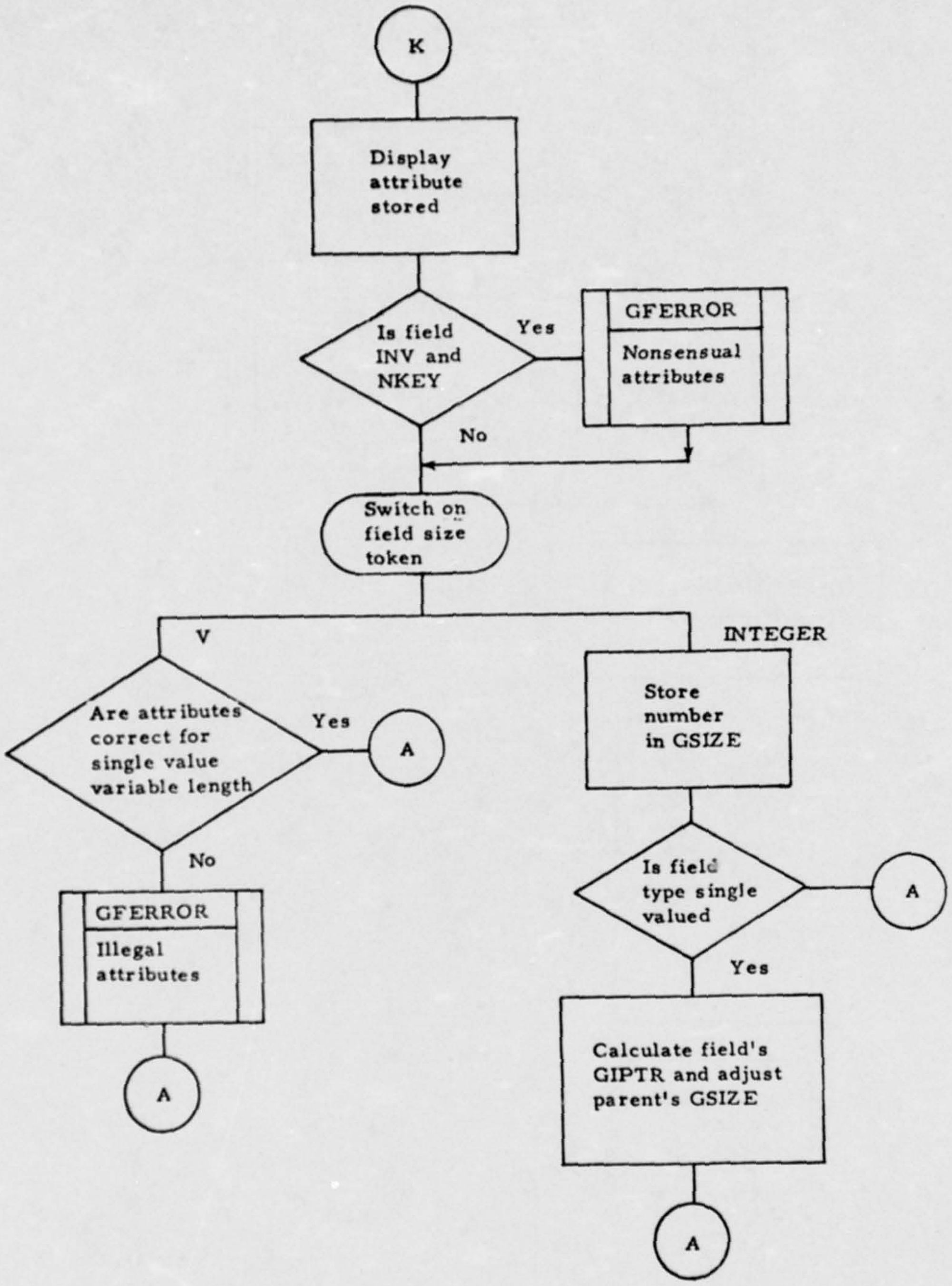

K
Display attribute stored
Is field INV and NKEY
Yes
GFERROR
Nonsensual attributes
No
Switch on field size token
V
INTEGER
Are attributes correct for single value variable length
Yes
A
No
GFERROR
Illegal attributes
A
Store number in GSIZE
Is field type single valued
A
Yes
Calculate field's GIPTR and adjust parent's GSIZE
A

Figure 39. DDL Process Data Flow (Sheet 5 of 9).

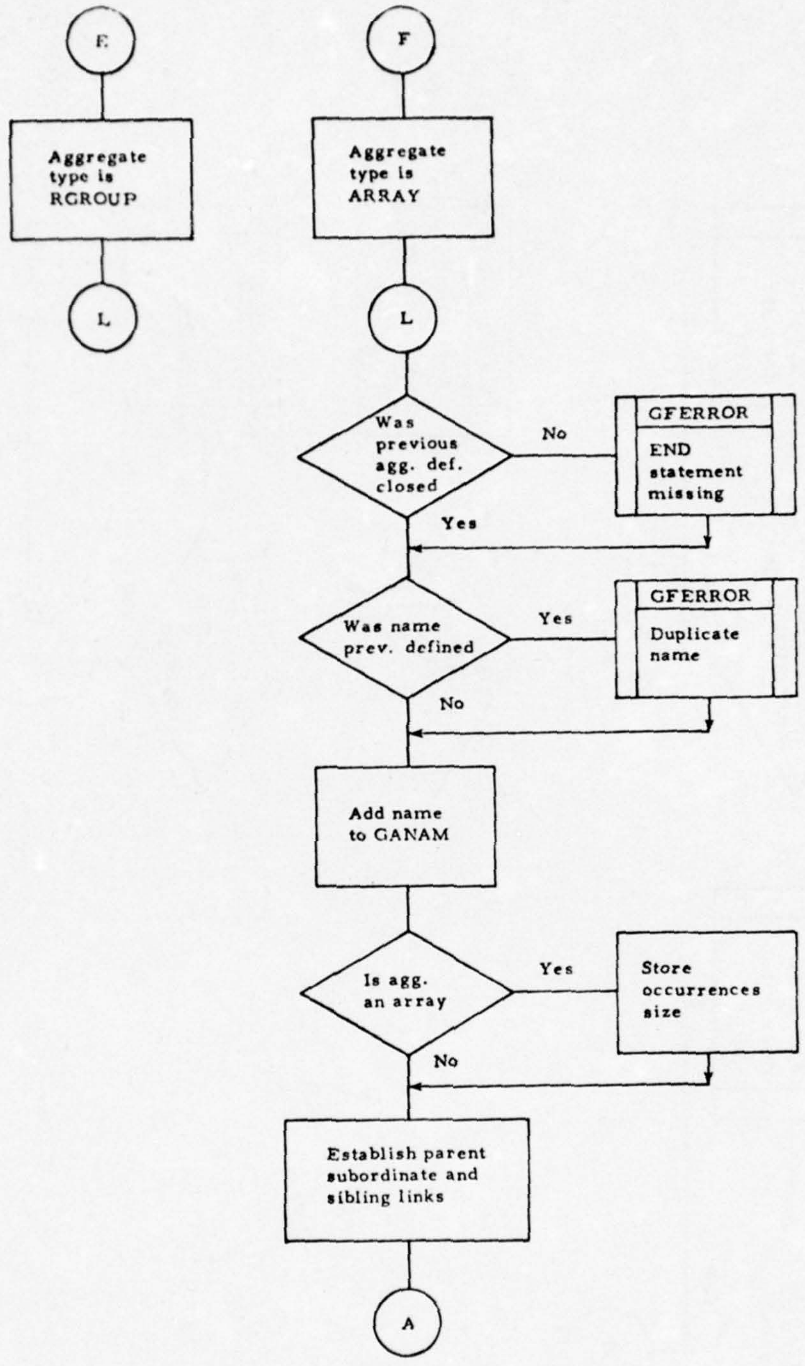


Figure 39. DDL Process Data Flow (Sheet 6 of 9).

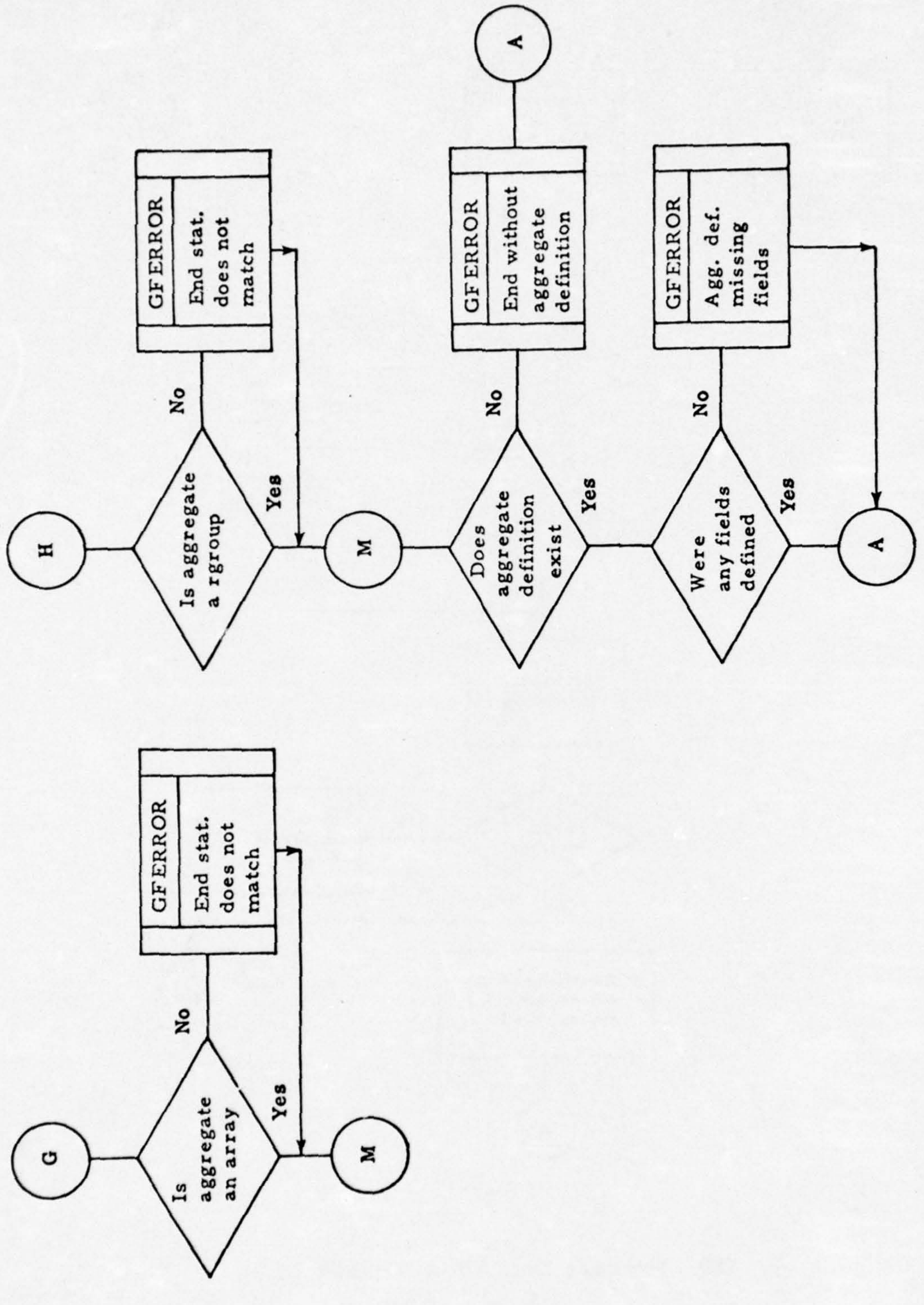


Figure 39. DDL Process Data Flow (Sheet 7 of 9).

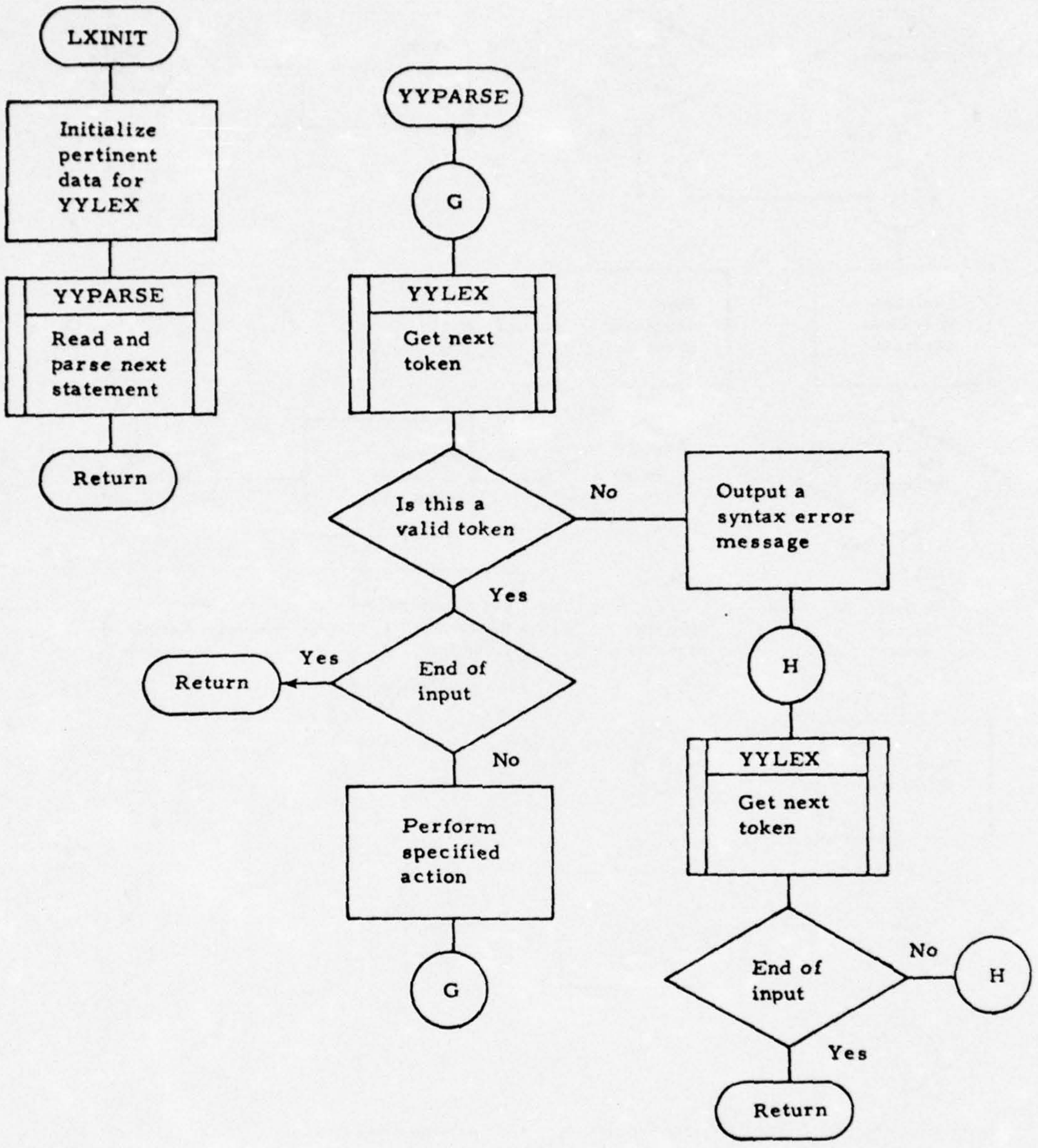


Figure 39. DDL Process Data Flow (Sheet 8 of 9).

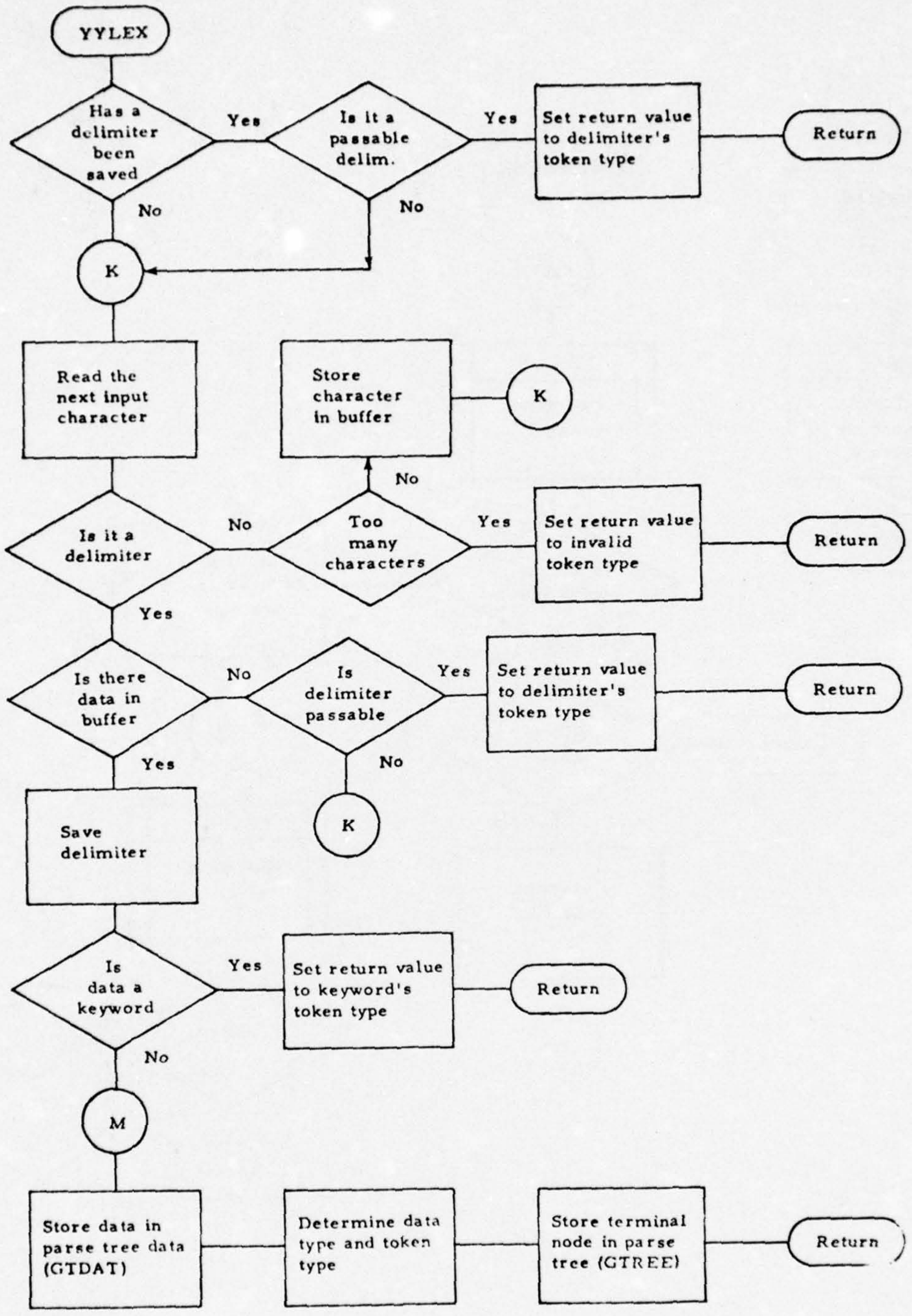

YYLEX
Has a delimiter been saved
Yes
Is it a passable delim.
Yes
Set return value to delimiter's token type
Return
No
No
K
Read the next input character
Store character in buffer
K
No
Is it a delimiter
No
Too many characters
Yes
Set return value to invalid token type
Return
Yes
Is there data in buffer
No
Is delimiter passable
Yes
Set return value to delimiter's token type
Return
No
K
Yes
Save delimiter
Is data a keyword
Yes
Set return value to keyword's token type
Return
No
M
Store data in parse tree data (GTDAT)
Determine data type and token type
Store terminal node in parse tree (GTREE)
Return

Figure 39. DDL Process Data Flow (Sheet 9 of 9).

YYPARSE (Parser) – YYPARSE is a parser which is produced by the Yet Another Compiler-Compiler (YACC) program running under UNIX. YACC produces the parser as well as a set of tables which the parser uses to organize the tokens passed to it by the lexical analyzer (YYLEX). These tables reflect the syntax of the DDL statements. YYPARSE calls YYLEX, which returns a value called a token type. If the token type is invalid according to the input statement syntax rules, an error message is output and YYPARSE calls YYLEX continuously until an end-of-input token type is returned. YYPARSE then returns with the return flag set to reflect a syntax error. If the token type is valid according to the input rules, the action specified in the YACC run is performed. The actions that can be invoked are: no action, build a parse tree node with the specified number of legs, store the final token type and write out the parse tree and parse tree data. After the action is performed, YYPARSE calls YYLEX for the next token type. When the end-of-input token type is received and its action performed, YYPARSE returns with the return flag set to reflect statement acceptance. If YYLEX encounters an EOF, it returns an end-of-input token type and sets the return flag to reflect EOF.

YYLEX (Lexical Analyzer) – YYLEX is the function which actually inputs and processes each DDL statement. YYLEX reads and saves each input character until a delimiter is enountered. The delimiters for ADAPT I DDL are: comma, newline (carriage return), space, left parenthesis, and right parenthesis. If an EOF is encountered, the return flag is set to reflect EOF and an end-of-input token type is returned. If a delimiter is encountered by itself, it is either passed to the parser (comma, left or right parenthesis, or newline), or it is ignored. When a delimiter terminates a string of characters, YYLEX must determine what the characters represent. If the characters form a DDL keyword, YYLEX returns with that keyword's token type. Otherwise, YYLEX determines the type of constant which is formed by the characters (name, integer), stores the value of the constant in the parse tree data (GTDAT), stores a terminal node in the parse tree (GTREE), and returns with the constant's token type. If YYLEX encounters an error (e. g. , too many characters input), it returns an invalid token type to the parser and thereby generates a syntax error message.

TSIGI (Target System Input Generator: Interactive)

The TSIGI process is the ADAPT I process which controls the conversion of UDL statements into appropriate "interactive" target system transformation strings. This process provides both of the functions performed by TSIGB and NETRES processes for "batch" interrogations. The primary difference between "interactive" and "batch" interrogations is that a user is essentially "connected" to an interactive host system when referencing and receiving data from some file and, in a batch system, the acts of interrogation and the reception of subsequent responses are two separate operations which may sometimes be separated by a lengthy interval.

GLOBAL DATA USAGE – TSIGI utilizes the following global data:

GLABL – statement label file. TSIGI inserts and/or deletes user statement labels from this file. Only labels on a FIND statement are applicable.

GLIST – user list file. TSIGI inserts and deletes user-specified lists (explicit lists) as well as implicit lists from this file.

GRMAP – record map file. TSIGI creates this file as a receptacle for converted response strings.

GRDAT – record data file. TSIGI creates this file as a receptacle for converted response strings.

GTREE – parse tree. TSIGI uses the parse tree for normal UDL statement/command processing.

GTDAT – parse tree data. TSIGI uses the parse tree data for normal UDL statement/command processing.

GFILD – file description. TSIGI uses this file to ascertain the target system that is being queried.

LOCAL DATA USAGE – All three local data structures, TITRG, TIPRM, and TIACT, which are utilized by TIDECOMP, contain target system-specific information with respect to query language constructs.

TIPRM – used by TIDECOMP to point to appropriate "transformation string" constructs in TITRG and to appropriate "action functions" in TIACT. TIPRM is indexed by UDL tokens contained in GTREE.

TITRG – contains actual transformation string constructs which represent the UDL token in the target system query language. TITRG is indexed by TIPRM.

TIACT – contains "switch" pointers to appropriate "action functions" and parse tree "leg indicators" for function arguments. TIACT is pointed to by TIPRM and its ordered access is activated by "escape characters" in the transformation string constructs in TITRG.

GENERAL PROCESS FLOW – TSIGI is activated by EXEC. It is called when EXEC detects an OPEN, CLOSE, FIND, DISPLAY, or SAVE statement that is applicable for an interactive target system; i.e., the referenced file is resident on a distant host which is "interactive" in nature. TSIGI reads in both parse tree files (GTREE and GTDAT) and the file description table (GFILD). GFILD is used to ascertain the applicable target system which is required in order to set up pointers to the correct decompiler tables. TSIGI now branches on the statement/command token type. If the statement is an OPEN statement, TSIGI retrieves the indicated file-identifiers (FLIDs) from the parse tree (GTREE) and inserts them into the "open-file" list. If a CLOSE statement is encountered, TSIGI removes the indicated FLIDs from the "open-file" list.

The occurrence of a FIND statement causes TSIGI to call function TIADDLAB in order to save the statement label. TIADDLAB saves the label in GLABL and establishes links to the parse tree tables (GTREE and GTDAT) representing the FIND statement. Following this, TSIGI creates a transformation string file (c000) for the decompiler.

TSIGI then calls the decompiler, TIDECOMP. TIDECOMP is responsible for generating the appropriate transformation strings which represent the UDL FIND statement as a target system query. After TIDECOMP generates the transformation strings, TSIGI writes them out to the network via the NCP daemon. TSIGI then initiates a network read. Upon receiving data from the network, TSIGI analyzes the responses. If the query was successful, TSIGI determines the number of records satisfying the query and outputs the result to the user. If the query was not successful (i. e., due to semantical errors, hardware errors, etc.), an appropriate diagnostic is output to the user.

If a DISPLAY statement is encountered, TSIGI checks to see if a list is to be displayed. If this is the case, transformations are not necessary, and TSIGI must only ascertain the list-name from GTDAT, open the two record files (GRMAP and GRDAT), and call the DISPLAY process. However, if the DISPLAY statement references a FIND statement (via a statement "tag"), TSIGI must call the TILIST function to establish an "implicit" list for the user. The list is associated with the record data received from the target system. TSIGI then calls the decompiler (TIDECOMP) which generates the appropriate transformation strings in file c000. When this is completed, TSIGI writes the transformation strings to the network and then initiates a read for the response. The Target System Output Translator (TSOT) process is called to translate the target system record data responses into UDL records. That is, the responses are translated and then placed into a set of GRMAP and GRDAT files. The implicit list created by TILIST is associated with these data. TSIGI now calls the DISPLAY process, passing to it the DISPLAY statement parse trees (GTREE and GTDAT) and the implicit list record data (GRMAP and GRDAT). Upon return from DISPLAY, TSIGI deletes the record data (GRMAP and GRDAT) and removes the implicit list-name from the user's list file, GLIST.

If TSIGI encounters a SAVE command, TILIST is called to create an explicit list and TIDECOMP is called to generate the appropriate transformation strings in file c000. As was the case with the DISPLAY statement, TSIGI writes the transformation strings to the network (via the NCP daemon) and initiates a read for the subsequent response. Upon receipt of the target system responses, TSIGI calls TSOT in order to translate the response strings into proper UDL record data. Since the user initiated a SAVE command, the record data contained in GRMAP and GRDAT are associated with the user-specified list-name and are saved until the user deletes them (via the DELETE command). Figure 40 provides more information on the TSIGI data flow.

MAJOR FUNCTION DESCRIPTIONS –

TILIST (List Generator) – TILIST controls the generation of a list in the user's list file. TILIST is passed a parameter which specifies the type of list to be generated. If an implicit list is requested, TILIST uses "a000" as a default name and inserts it into the user's list file GLIST. If an explicit list is asked for, TILIST retrieves the user-specified list-name from GTDAT and inserts it into GLIST. In either situation, the list-name is used to create record data files, GRMAP and GRDAT. These files are associated with the list-name and will contain target system record data (converted to UDL format) from a subsequent transformation process.

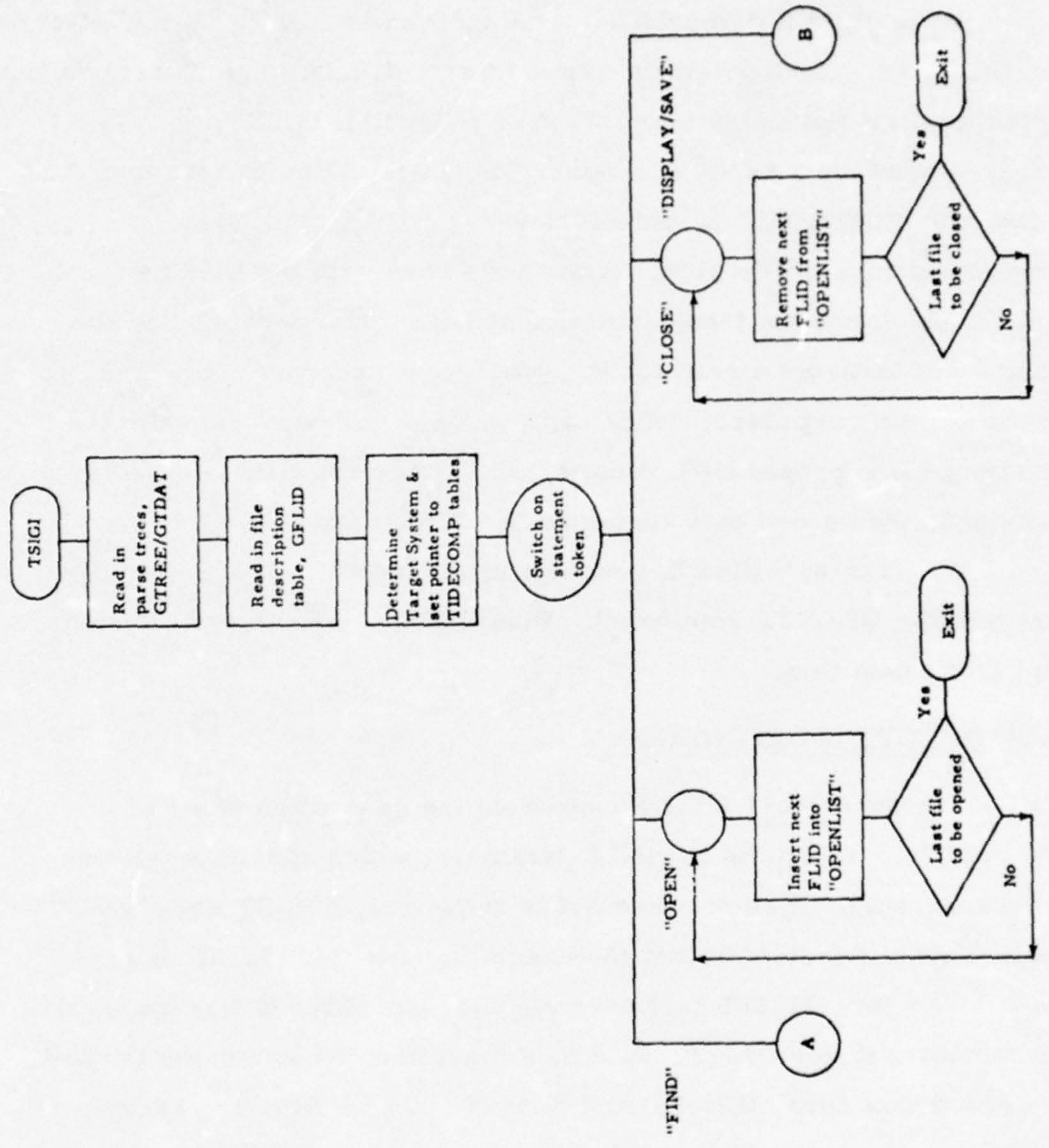

TSIGI
Read in parse trees, GTREE/GTDAT
Read in file description table, GFLID
Determine Target System & set pointer to TIDECOMP tables
Switch on statement token
"FIND"
A
"OPEN"
Insert next FLID into "OPENLIST"
Last file to be opened
Yes
No
Exit
"CLOSE"
Remove next FLID from "OPENLIST"
Last file to be closed
Yes
No
Exit
"DISPLAY/SAVE"
B

Figure 40. TSIGI Process Data Flow (Sheet 1 of 5).

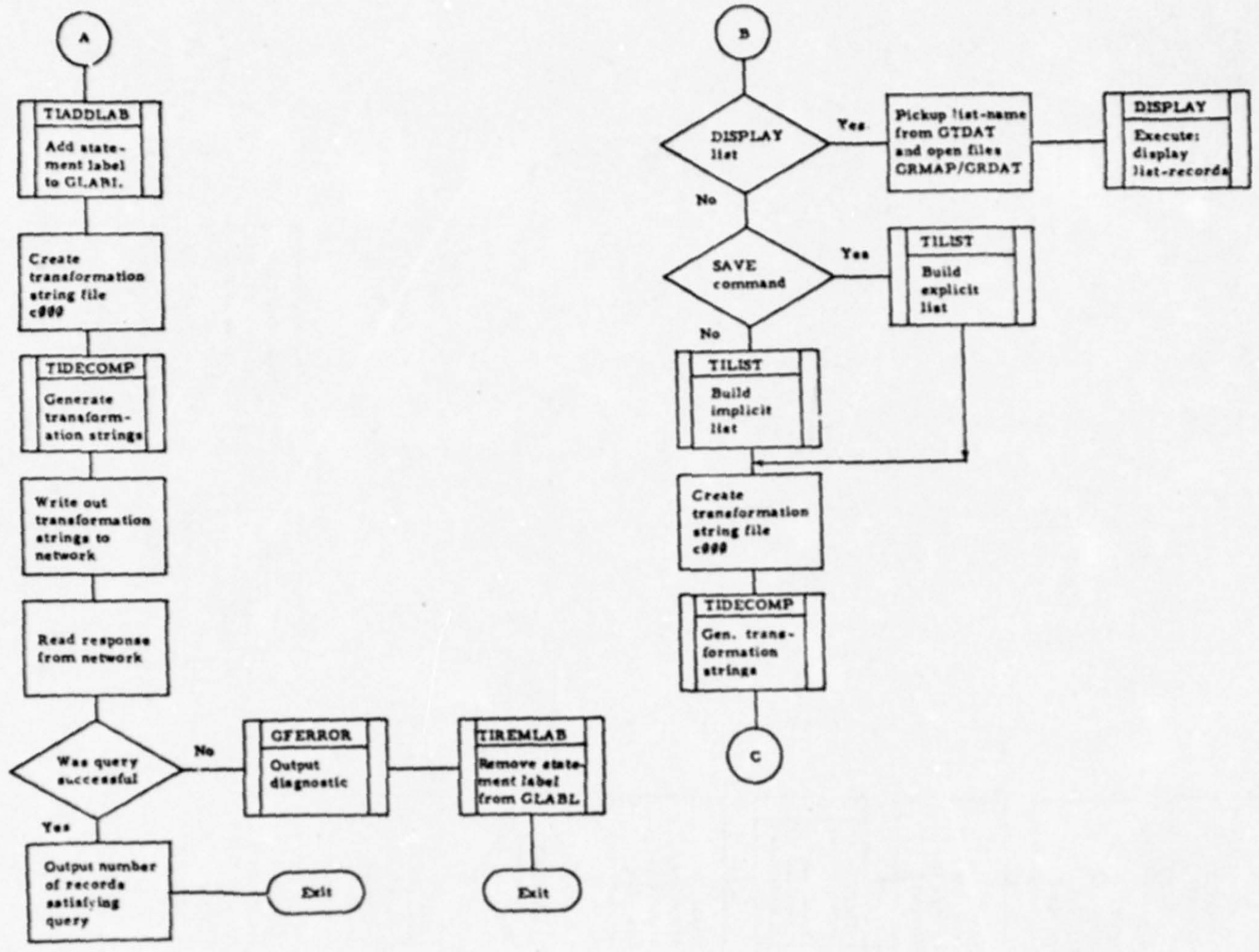

A
TIADDLAB
Add statement label to GLABL
Create transformation string file c000
TIDECOMP
Generate transformation strings
Write out transformation strings to network
Read response from network
Was query successful
No
Yes
GFERROR
Output diagnostic
TIREMLAB
Remove statement label from GLABL
Output number of records satisfying query
Exit
Exit
B
DISPLAY list
Yes
Pickup list-name from GTDAT and open files GRMAP/GRDAT
DISPLAY
Execute: display list-records
No
SAVE command
Yes
TILIST
Build explicit list
No
TILIST
Build implicit list
Create transformation string file c000
TIDECOMP
Gen. transformation strings
C

Figure 40. TSIGI Process Data Flow (Sheet 2 of 5).

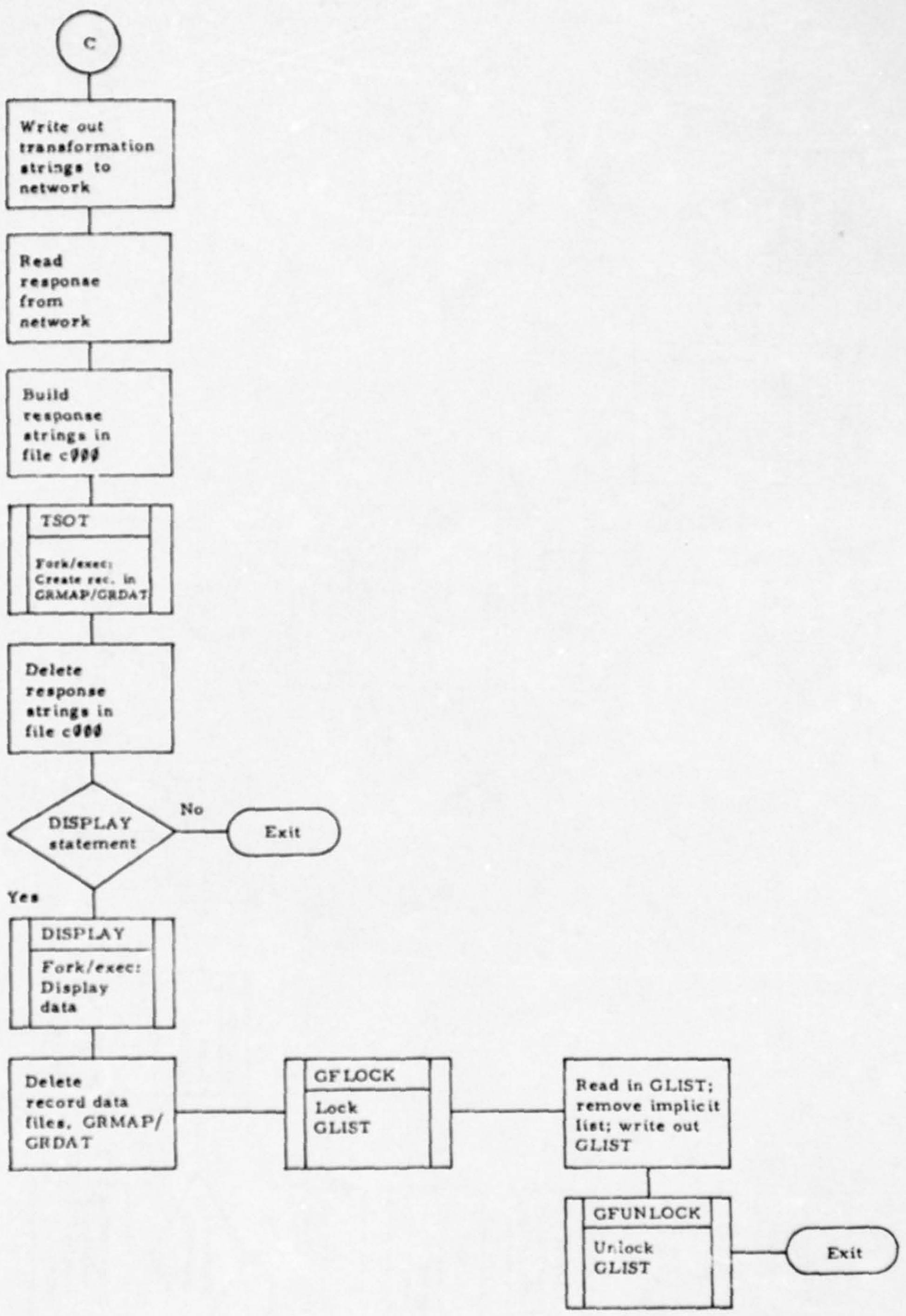

C
Write out transformation strings to network
Read response from network
Build response strings in file c000
TSOT
Fork/exec: Create rec. in GRMAP/GRDAT
Delete response strings in file c000
DISPLAY statement
No
Exit
Yes
DISPLAY
Fork/exec: Display data
Delete record data files, GRMAP/GRDAT
GFLOCK
Lock GLIST
Read in GLIST; remove implicit list; write out GLIST
GFUNLOCK
Unlock GLIST
Exit

Figure 40. TSIGI Process Data Flow (Sheet 3 of 5).

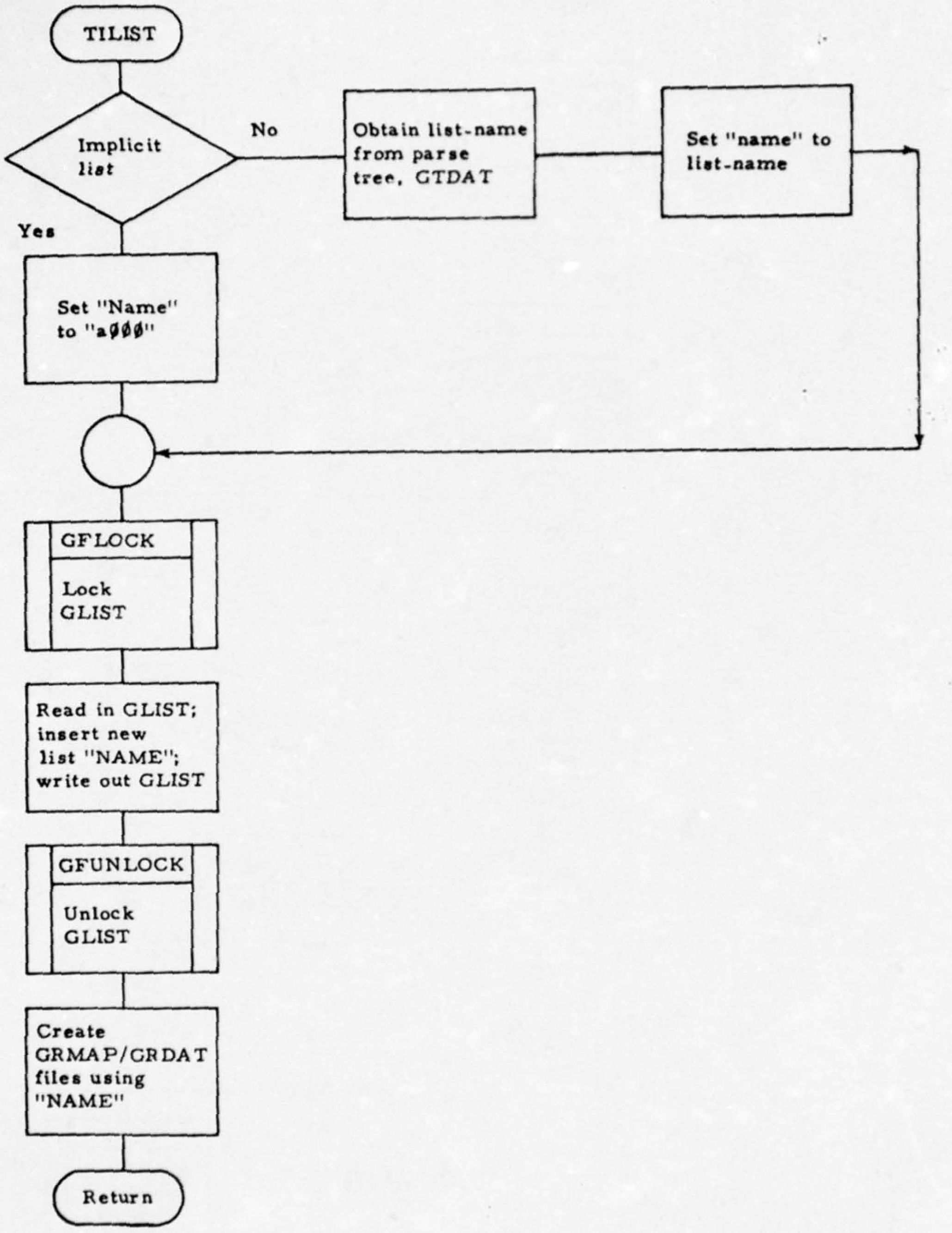

TILIST
Implicit list
No
Yes
Obtain list-name from parse tree, GTDAT
Set "name" to list-name
Set "Name" to "aØØØ"
GFLOCK
Lock GLIST
Read in GLIST; insert new list "NAME"; write out GLIST
GFUNLOCK
Unlock GLIST
Create GRMAP/GRDAT files using "NAME"
Return

Figure 40. TSIGI Process Data Flow (Sheet 4 of 5).

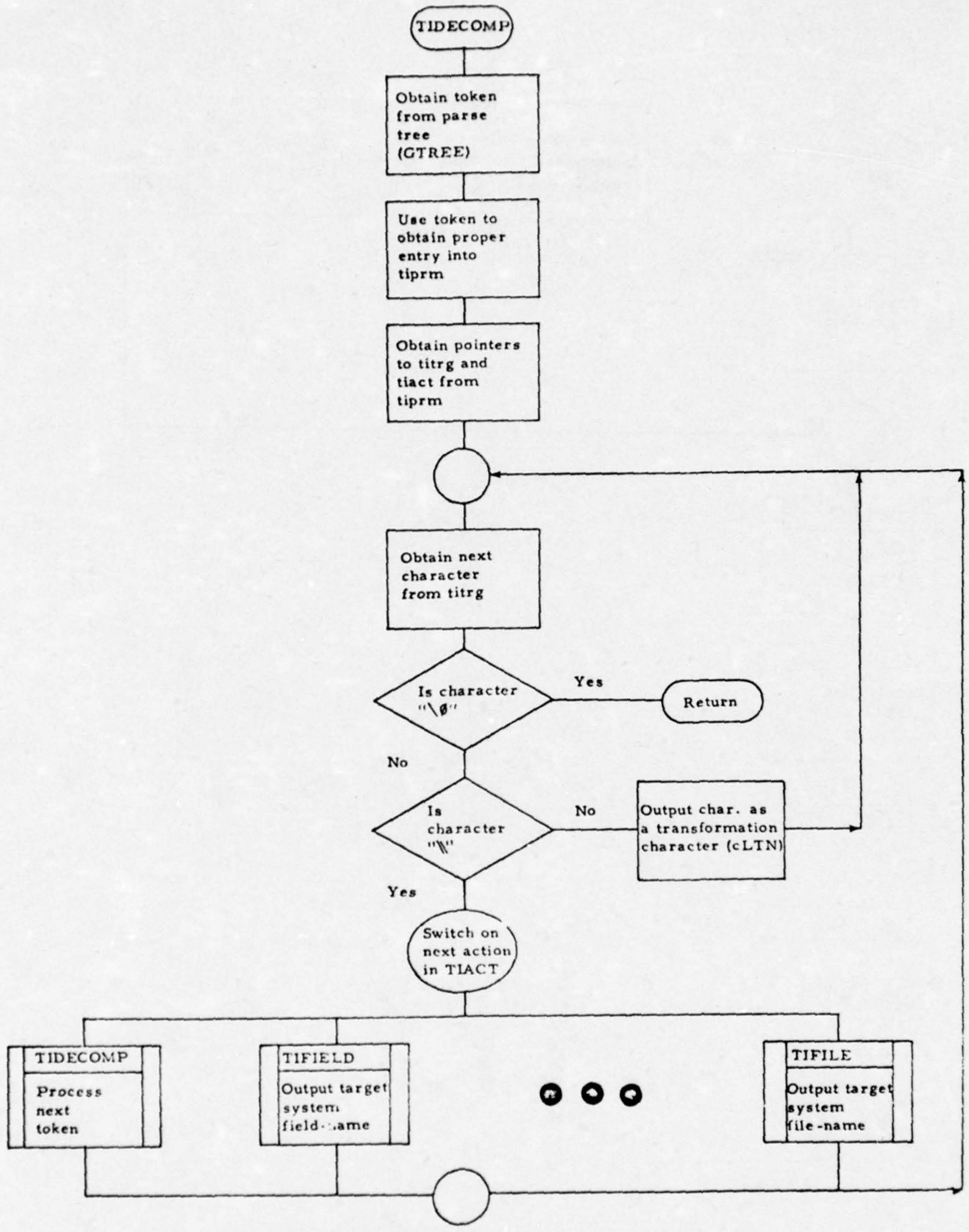

TIDECOMP
Obtain token from parse tree (GTREE)
Use token to obtain proper entry into tiprm
Obtain pointers to titrg and tiact from tiprm
Obtain next character from titrg
Is character "\Ø"
Yes
Return
No
Is character "\\"
No
Output char. as a transformation character (cLTN)
Yes
Switch on next action in TIACT
TIDECOMP
Process next token
TIFIELD
Output target system field-name
TIFILE
Output target system file-name

Figure 40. TSIGI Process Data Flow (Sheet 5 of 5).

TIDECOMP (Decompiler) – The decompiler, TIDECOMP, is responsible for the generation of transformation strings for the various target systems across the network. TIDECOMP, when called, is passed a pointer into a parse tree, GTREE. Usually this pointer references the statement/command token at the "head" of the tree, although this is not a requirement. TIDECOMP uses the primitive UDL token as an index into local table TIPRM. TIPRM points into two other tables: transformation string table TITRG and "action" table TIACT. The pointer to TITRG points to a set of characters representing the UDL token for that target system. The pointer to TIACT points to a set of "action" indicators that are required to satisfy the transformation of the UDL token. The various "actions" are triggered by TIDECOMP control characters in TITRG. From this point on, TITRG controls the actions of TIDECOMP.

Each character in TITRG (starting with the character pointed to initially by TIPRM) is analyzed to see if it is one of two TIDECOMP control characters: "\0" and "\\." The null character "\0" indicates the end of the transformation string for a UDL token and the back-slash character "\\" indicates the execution of an "action" function. If the character is not a control character, it is output as a transformation string character. The null character causes TIDECOMP to return. The back-slash character causes TIDECOMP to perform the next "action" contained in TIACT (starting initially with the action pointed to by TIPRM). TIACT also specifies an optional relative parse tree leg pointer for input to the "action." The majority of "action" functions utilized by TIDECOMP are general in nature and hence are used for most target system transformations. Following is a short description of some of these functions:

a. Continue. This function is passed a pointer into the parse tree, GTREE. It calls TIDECOMP passing the GTREE pointer as input. This function is used to transverse the parse tree.

b. Field-name output. This function is passed a pointer into the parse tree table, GTREE. GTREE, at this point, contains the UDL field identifier

(FID) for the reference field. The FID is used as an index into the appropriate field transformation name table. The target system field-name is output as a transformation string.

c. File-name output. This function is passed a pointer into the parse tree table, GTREE. GTREE, at this point, contains the UDL file identifier (FLID). The FLID is used as an index into the file transformation name table. The target system file-name is output as a transformation string.

d. Data-constant output. This function is passed a pointer into the parse tree data table, GTDAT. GTDAT, at this point, contains the referenced data-constant which is output as a transformation string.

e. TS-name table. This function is passed a file-identifier (FLID). The appropriate set of target system file and field name tables are read in by this function. These tables are utilized by other TIDECOMP functions.

f. File-name retrieve. This function is passed a pointer into the parse tree, GTREE. GTREE, at this point, contains a file-identifier (FLID). The TS-name table function is now called with the FLID as input.

g. Tag retrieve. This function is passed a pointer into the parse tree data table, GTDAT. GTDAT, at this point, contains a UDL statement tag. The user label table (GLABL) is read in and searched for the tag. The file identifier (FLID) is then picked up from GLABL and used as a parameter to the TS-name table function.

h. Dependent interrogation. This function is passed a pointer into the parse tree data table, GTDAT. GTDAT, at this point, contains a UDL statement tag. The tag is used to generate path names to the referenced FIND statement parse trees (GTREE/GTDAT). These files are opened and read in.

i. Write-to-network. This function writes out the current set of transformation strings to the network and then resets the transformation string file to zero.

j. Read-from-network. This function reads from the network and then compares the result with the characters currently residing in the transformation string file. If a match is made, the transformation string file is reset to zero. If a match is not made, another read is initiated and the compare is tried again. This function and the "write-to-network" function are used to initiate "interactions" with the various interactive distant hosts. The match is made against anticipated system responses.

TSIGB (Target System Input Generator:Batch)

The TSIGB process is the ADAPT I process which controls all conversions of UDL statements into appropriate batch target system transformation strings. This process plus NETRES performs analogous functions to those provided by TSIGI.

GLOBAL DATA USAGE – The following ADAPT I global data structures are utilized by the TSIGB process:

GTDES – transaction description file. GTDES contains descriptions of all batch transactions currently active for users under ADAPT I. This file is used by the TSIGB main function, the transaction initiation function (TILTIN), and the transaction transmit function (TITRLTN).

GLABL – user label description file. TSIGB inserts and/or deletes user labels from this file. Each logical batch transaction is represented by a FIND/DISPLAY or FIND/SAVE UDL statement/command pair and is represented by a label; i. e., the label on the FIND statement.

GUSER – user description file. Used by TSIGB to retrieve the next logical transaction number (LTN) available for this user. LTNs are local to a given user, their uniqueness across ADAPT I being determined by the "concatenated-value" of a users identifier index and the LTN.

GLIST – user list description file. Used for the declaration of both implicit and explicit lists for the user initiating the batch transaction. Implicit lists are declared temporarily for FIND/DISPLAY sequences and explicit lists are declared permanently.

GTREE – parse tree file. Used for normal UDL statement/command processing.

GTDAT – parse tree data file. Used for normal UDL statement/command processing.

LOCAL DATA USAGE – Refer to the discussion for this paragraph in TSIGI. Also refer to the description of the TSIGI decompiler, TIDECOMP.

GENERAL PROCESS FLOW – TSIGB is activated by EXEC. It is called when EXEC detects transformable UDL statement/command sequences for a batch-oriented target system; i.e., the TIPS/TILE system. TSIGB reads in both parse tree tables (GTREE and GTDAT) and switches on the statement/command token. Transformable statements/commands for batch transaction sequences (as well as "interactive" sequences) are FIND, DISPLAY, and SAVE. Each statement/command is validated for compliance to batch transaction operations. If the statement is a FIND statement (and assuming it has complied with batch transaction operations), TSIGB calls TIADDLAB to add the FIND statement label to GLABL. The reading/writing of GLABL is controlled by TIADDLAB. After the label has been processed, the transaction initiation function, TILTIN, is called to reserve a logical transaction number (LTN) for this user batch sequence. TSIGB then calls the decompiler, TIDECOMP, which generates "transformation strings" representing the UDL FIND statement in terms of the target system query language.

If the UDL statement is a DISPLAY statement, the TSIGB main function calls TIDECOMP to generate appropriate "transaction strings" which are concatenated to other "transformation strings" that exist in the transformation string file, cLTN. It must be remembered that UDL DISPLAY statements must reference a previous FIND statement, and, therefore, "transformation strings" for a previous FIND statement must already exist in cLTN. Following the generation of "transformation strings," the batch transaction is transmitted by TITRLTN. A logical transaction can be mapped onto a single FIND/DISPLAY sequence. The GTREE/GTDAT structure pair representing the DISPLAY statement is saved (linked to) for target system response processing by the DISPLAY process. The TSIGB processing for the

SAVE command is very similar to the DISPLAY statement except that an explicit list is generated for the transaction (as directed by the SAVE command) and GTREE/GTDAT parse tree structures are not saved. See figure 41 for more information.

MAJOR FUNCTION DESCRIPTIONS -

TILTIN (Transaction Initiation) - TILTIN reads in the transaction description table (GTDES) under a "locked condition" (i.e., read for update) and reserves the next available slot. GUSER is read in a "lock condition" and the next logical transaction number (LTN) is retrieved for the user. Using the LTN, TILTIN generates a user-unique path name, cLTN, for the transformation string file, which is then created.

The "string" file is used to hold temporary transformation information including both "transformation strings" and "response strings." This structure is used for the duration of the transaction up to and including the target system response.

TITRLTN (Transaction Transmit) - TITRLTN is called to generate either an implicit list or an explicit list. This list will be associated with the record data (GRMAP/GRDAT) representing the target system response (refer to NETRES process). Upon creation of the new list in GLIST and associated entries into the user's list directory, TITRLTN calls the Batch Network Control Program (BNCP) to generate the "INTG" message, using the "transformation strings" in cLTN, for transmission through the network. At this point, TITRLTN outputs the LTN to the user's terminal for later perusal of the response.

TIDECOMP (Decompiler) - Refer to the description of TIDECOMP in the TSIGI process.

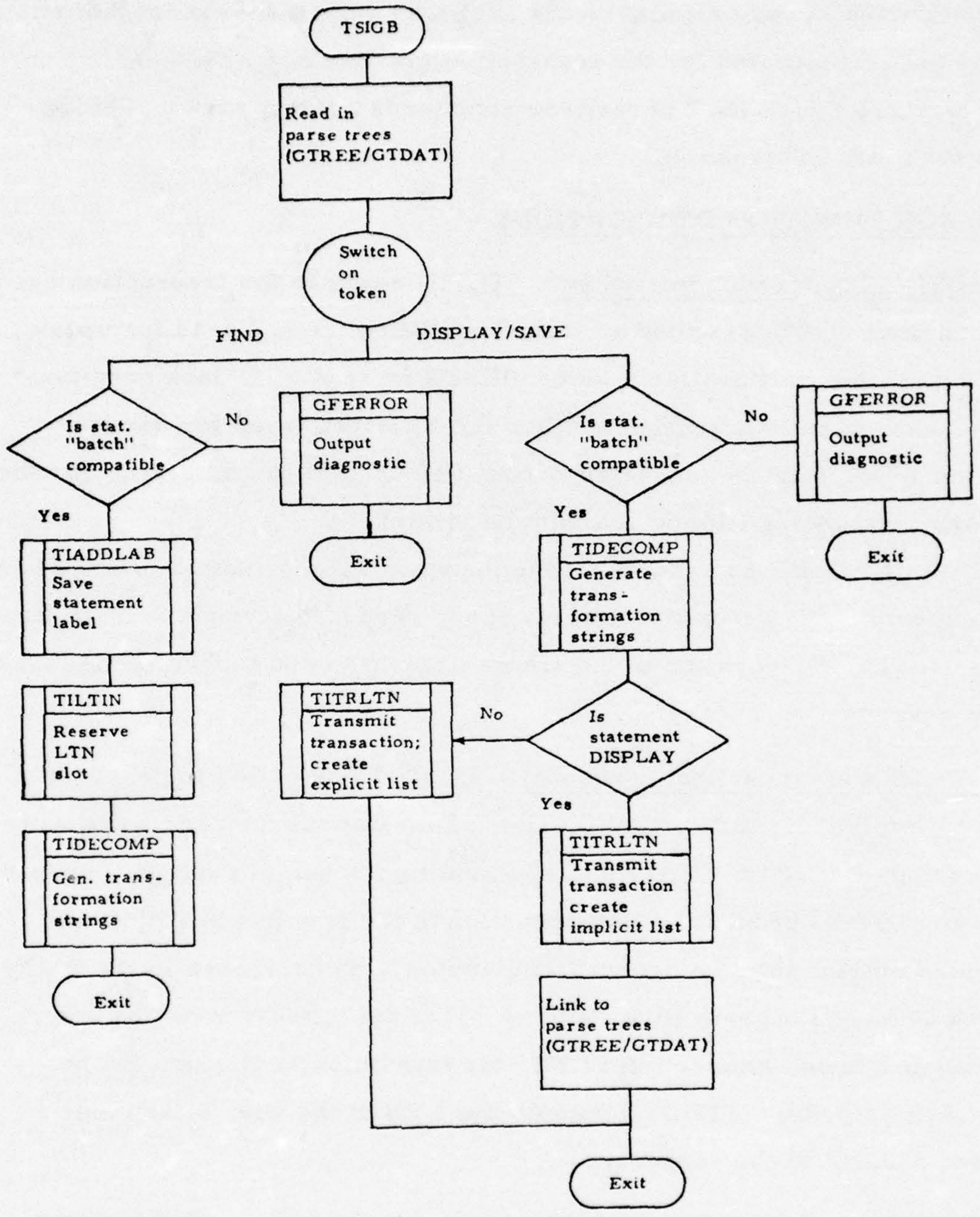

TSIGB
Read in parse trees (GTREE/GTDAT)
Switch on token
FIND
DISPLAY/SAVE
Is stat. "batch" compatible
No
Yes
GFERROR
Output diagnostic
Exit
TIADDLAB
Save statement label
TILTIN
Reserve LTN slot
TIDECOMP
Gen. trans-formation strings
Exit
Is stat. "batch" compatible
No
Yes
GFERROR
Output diagnostic
Exit
TIDECOMP
Generate trans-formation strings
Is statement DISPLAY
No
Yes
TITRLTN
Transmit transaction; create explicit list
TITRLTN
Transmit transaction create implicit list
Link to parse trees (GTREE/GTDAT)
Exit

Figure 41. TSIGB Process Data Flow (Sheet 1 of 2).

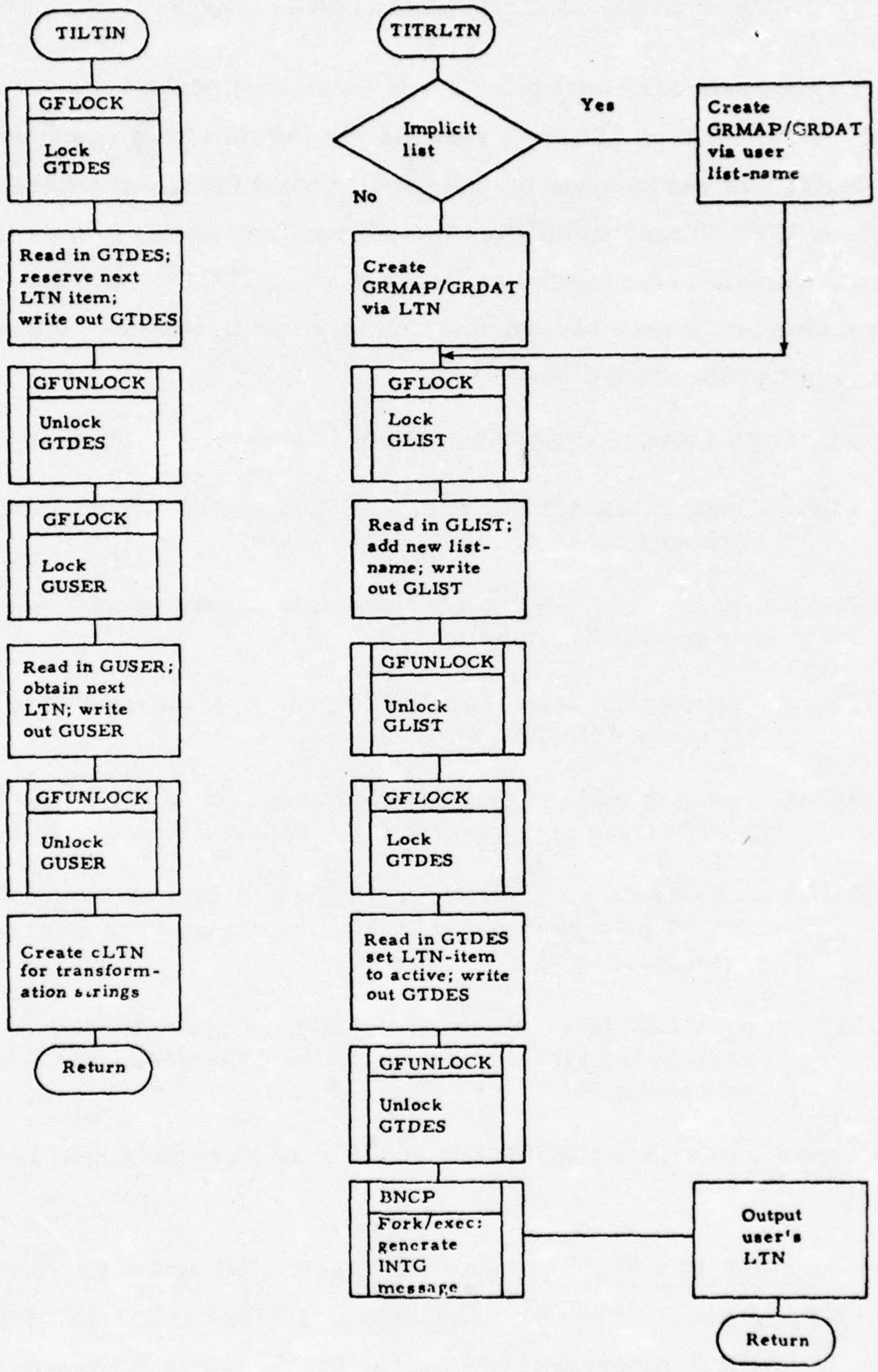

TILTIN
GFLOCK
Lock GTDES
Read in GTDES; reserve next LTN item; write out GTDES
GFUNLOCK
Unlock GTDES
GFLOCK
Lock GUSER
Read in GUSER; obtain next LTN; write out GUSER
GFUNLOCK
Unlock GUSER
Create cLTN for transformation strings
Return
TITRLTN
Implicit list
Yes
No
Create GRMAP/GRDAT via user list-name
Create GRMAP/GRDAT via LTN
GFLOCK
Lock GLIST
Read in GLIST; add new list-name; write out GLIST
GFUNLOCK
Unlock GLIST
GFLOCK
Lock GTDES
Read in GTDES set LTN-item to active; write out GTDES
GFUNLOCK
Unlock GTDES
BNCP
Fork/exec: generate INTG message
Output user's LTN
Return

Figure 41. TSIGB Process Data Flow (Sheet 2 of 2).

NETRES (Network Response Process: Batch)

The NETRES process constitutes the other half of ADAPT I's batch query processing. Where TSIGB is responsible for initiating batch transactions, NETRES is responsible for processing the subsequent transaction responses. NETRES' responsibilities include the processing of batch responses, maintenance of the logical transaction file (GTDES), and the removal of various temporary record data and implicit list-names that might exist as a result of the transaction.

GLOBAL DATA USAGE – NETRES utilizes the following global data:

GTDES – logical transaction file. NETRES sets the logical transaction's mode to a completed state.

GTREE – parse tree. NETRES opens this structure and passes it to the DISPLAY process.

GTDAT – parse tree data. NETRES opens this structure and passes it to the DISPLAY process.

GRMAP – record map. NETRES creates this data structure for TSOT as a receptacle for the converted response strings.

GRDAT – record data. NETRES creates this data structure for TSOT as a receptacle for the converted response strings.

GLIST – user list descriptions. NETRES utilizes this file to ascertain the type of lists with which the responses are to be associated.

<u>LOCAL DATA USAGE</u> – NETRES does not utilize any pertinent local data.

<u>GENERAL PROCESS FLOW</u> – NETRES is activated by the Batch Network Control Program (BNCP) when BNCP has received either an "ANSR" message or an "ABRT" message. Of course, the foregoing messages must have been caused by a previous "INTG" message initiated by the TSIGB

process (via BNCP). Upon receipt of the batch message, NETRES reads in the transaction description file (GTDES) and identifies the message with respect to its user-identifier (UID) and logical transaction number (LTN). NETRES creates record data files (GRMAP and GRDAT) for converted response strings, and a temporary output file (name "dLTN") for the final target system responses; i. e., DISPLAY data, error messages, etc. The message is then converted to a set of logical response strings and placed in file cLTN for further analysis by TSOT.

If the message is an "ANSR" message, NETRES activates the TSOT process to build record data in the newly created GRMAP/GRDAT files. When TSOT has completed its task, NETRES then checks to see if the logical transaction was initiated by a DISPLAY statement or a SAVE command. If a DISPLAY statement initiated the transaction, the DISPLAY process must be called to output the specified data to the temporary output file, dLTN. The DISPLAY process is passed the file descriptors of dLTN, GRMAP, GRDAT, and the DISPLAY statement parse trees, GTREE and GTDAT. Upon completion of the DISPLAY process, NETRES must remove the temporary record files (GRMAP and GRDAT), and the implicit list from the user's GLIST file. Finally, NETRES modifies the transaction description, GTDES, to indicate that this logical transaction has been completed.

If the message is an "ABRT" message, the appropriate error diagnostic is output to the temporary output, dLTN, and the appropriate files, as discussed previously, are deleted. GTDES is modified to indicate a complete transaction. See figure 42 for data flow.

Users at their discretion can peruse the status of their logical transactions by entering the VIEW command. If a particular transaction is completed, the user can view the output from the temporary output, dLTN. Once a user views a particular transaction, the temporary output is deleted.

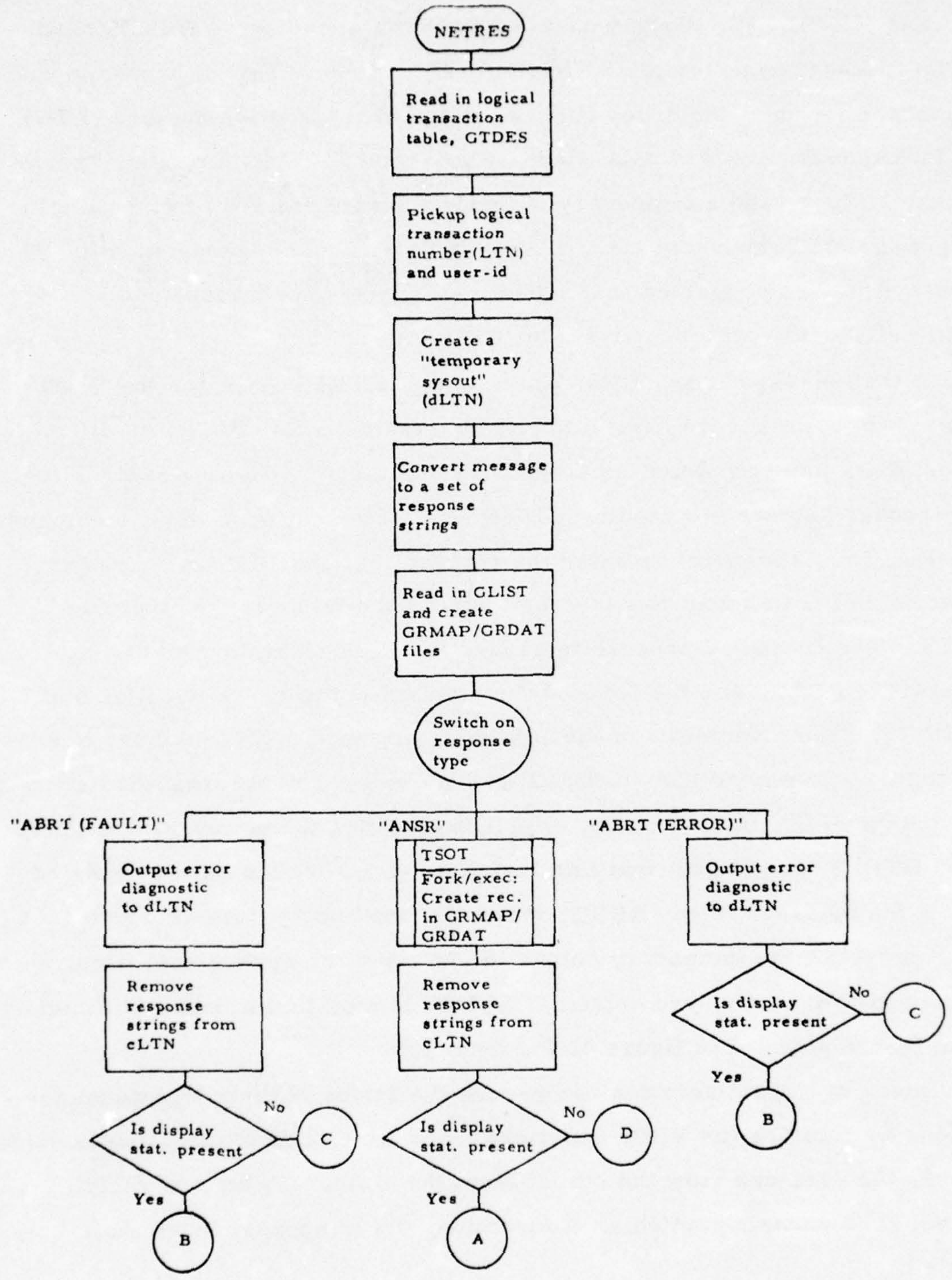

NETRES
Read in logical transaction table, GTDES
Pickup logical transaction number(LTN) and user-id
Create a "temporary sysout" (dLTN)
Convert message to a set of response strings
Read in GLIST and create GRMAP/GRDAT files
Switch on response type
"ABRT (FAULT)"
"ANSR"
"ABRT (ERROR)"
Output error diagnostic to dLTN
TSOT
Fork/exec: Create rec. in GRMAP/ GRDAT
Output error diagnostic to dLTN
Remove response strings from cLTN
Remove response strings from cLTN
Is display stat. present
No
C
Yes
B
Is display stat. present
No
C
Yes
B
Is display stat. present
No
D
Yes
A

Figure 42. NETRES Process Data Flow (Sheet 1 of 2).

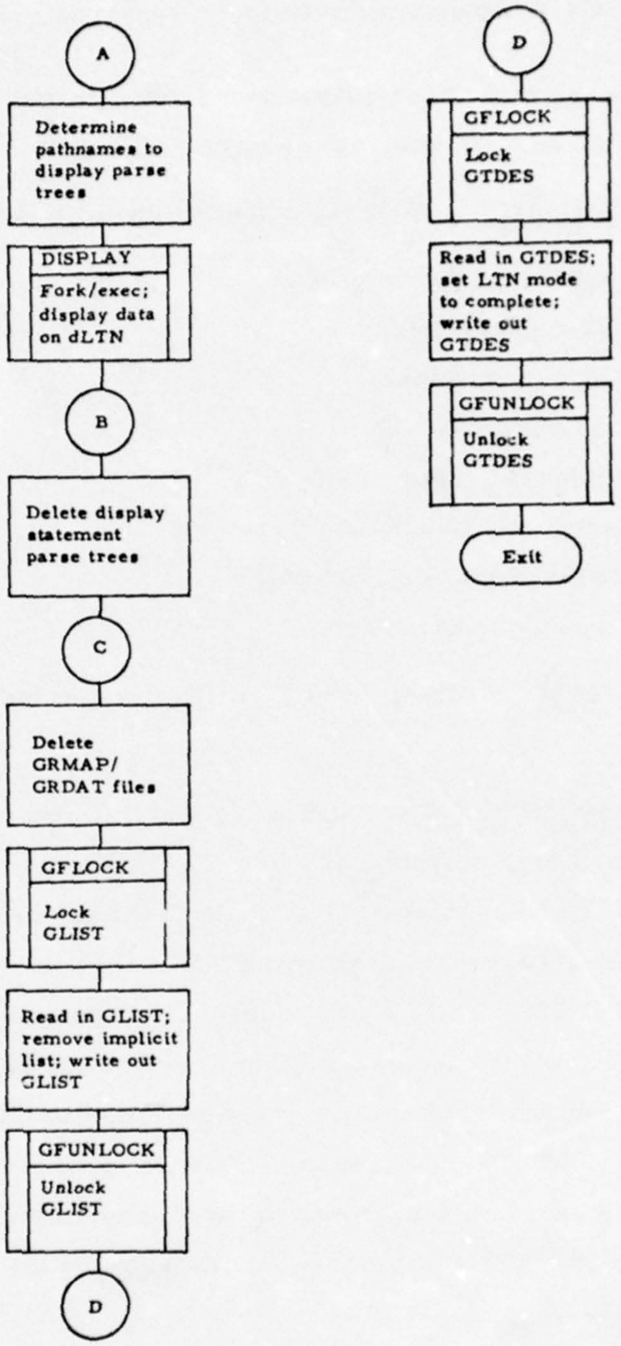

A
Determine pathnames to display parse trees
DISPLAY
Fork/exec; display data on dLTN
B
Delete display statement parse trees
C
Delete GRMAP/ GRDAT files
GFLOCK
Lock GLIST
Read in GLIST; remove implicit list; write out GLIST
GFUNLOCK
Unlock GLIST
D
D
GFLOCK
Lock GTDES
Read in GTDES; set LTN mode to complete; write out GTDES
GFUNLOCK
Unlock GTDES
Exit

Figure 42. NETRES Process Data Flow (Sheet 2 of 2).

TSOT (Target System Output Translator)

The TSOT process translates output data from a target system and converts the data into ADAPT internal records.

GLOBAL DATA USAGE – TSOT uses the following global data:

GADES – aggregate descriptions.

GFDES – field descriptions.

GFILD – file descriptions.

GFILN – file names.

GRMAP – internal record map.

GRDAT – internal record data.

GTFDS – transformation field data.

GMPHD – record map header.

LOCAL DATA USAGE – TSOT uses TOBUFF – input buffer for response strings.

GENERAL PROCESS FLOW – TSOT is activated whenever a user has requested data from a target system file and those data have been received. First, TSOT reads the file names (GFILN), file descriptions (GFILD), field descriptions (GFDES), aggregate descriptions (GADES), and the transformation field data (GTFDS). TSOT reads response strings and examines each string of characters until it recognizes the start of actual data. For a file whose data are recognized by character position, function TODATA is called to process the data. Otherwise, function TOFNAM is called to process the data for a file whose data are recognized by field name matching. When there are no more data, TSOT returns. Figure 43 provides more information on TSOT data flow.

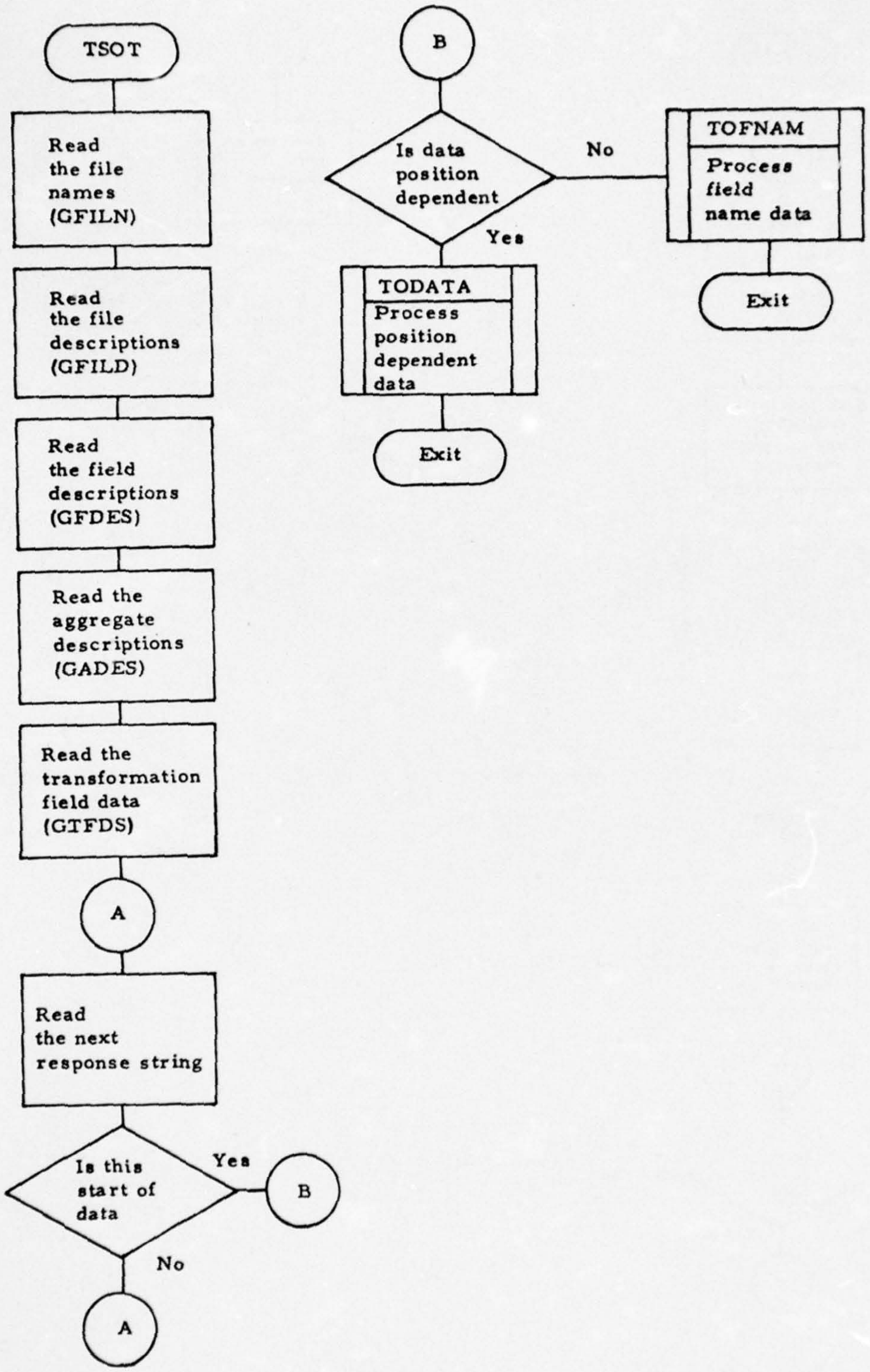

TSOT
Read the file names (GFILN)
Read the file descriptions (GFILD)
Read the field descriptions (GFDES)
Read the aggregate descriptions (GADES)
Read the transformation field data (GTFDS)
A
Read the next response string
Is this start of data
Yes
B
No
A
B
Is data position dependent
No
TOFNAM
Process field name data
Exit
Yes
TODATA
Process position dependent data
Exit

Figure 43. TSOT Process Data Flow (Sheet 1 of 4).

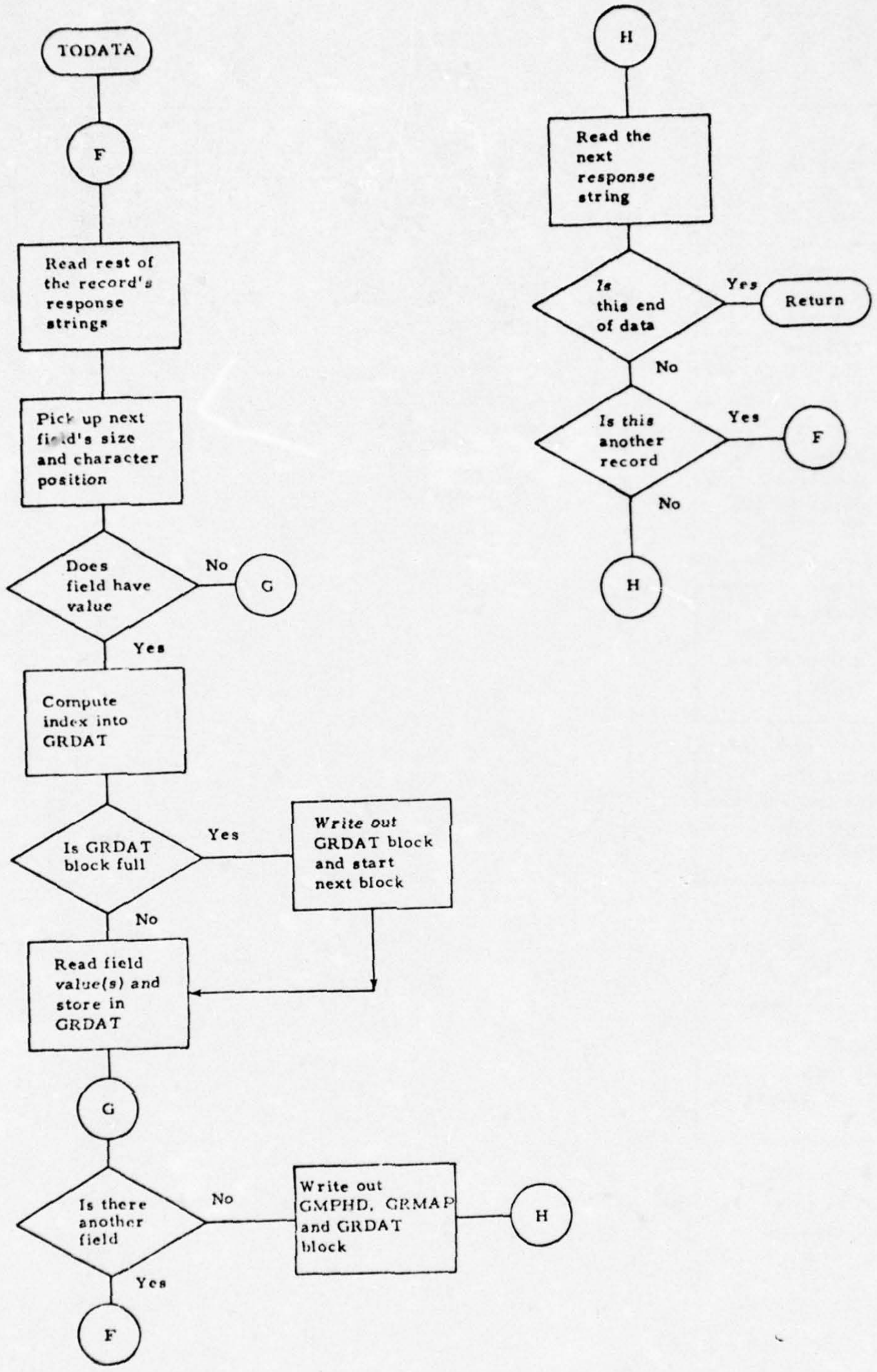

TODATA
F
Read rest of the record's response strings
Pick up next field's size and character position
Does field have value
No
G
Yes
Compute index into GRDAT
Is GRDAT block full
Yes
Write out GRDAT block and start next block
No
Read field value(s) and store in GRDAT
G
Is there another field
No
Write out GMPHD, GRMAP and GRDAT block
H
Yes
F
H
Read the next response string
Is this end of data
Yes
Return
No
Is this another record
Yes
F
No
H

Figure 43. TSOT Process Data Flow (Sheet 2 of 4).

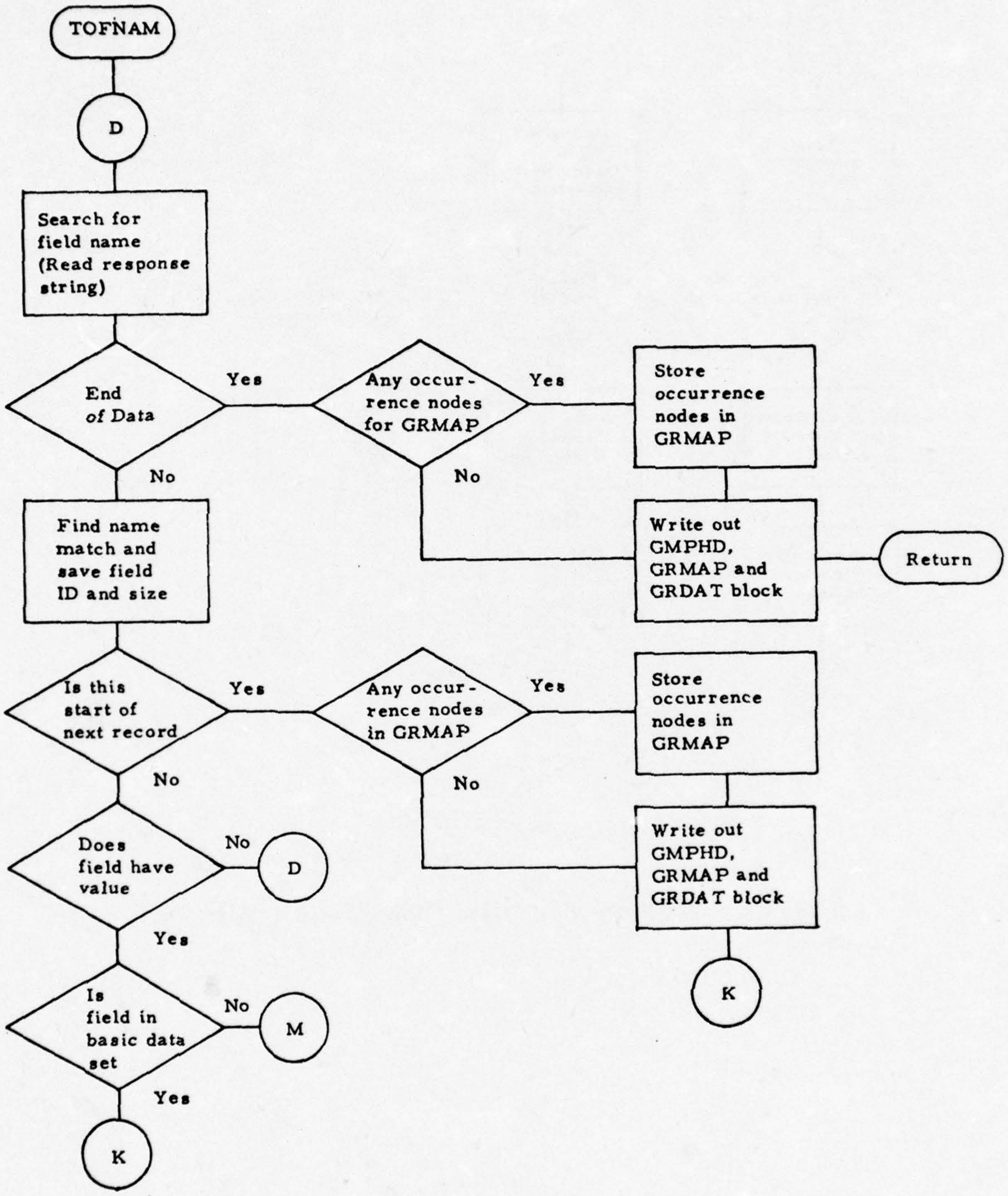

TOFNAM
D
Search for field name (Read response string)
End of Data
Yes
No
Any occurrence nodes for GRMAP
Yes
No
Store occurrence nodes in GRMAP
Write out GMPHD, GRMAP and GRDAT block
Return
Find name match and save field ID and size
Is this start of next record
Yes
No
Any occurrence nodes in GRMAP
Yes
No
Store occurrence nodes in GRMAP
Write out GMPHD, GRMAP and GRDAT block
K
Does field have value
No
D
Yes
Is field in basic data set
No
M
Yes
K

Figure 43. TSOT Process Data Flow (Sheet 3 of 4).

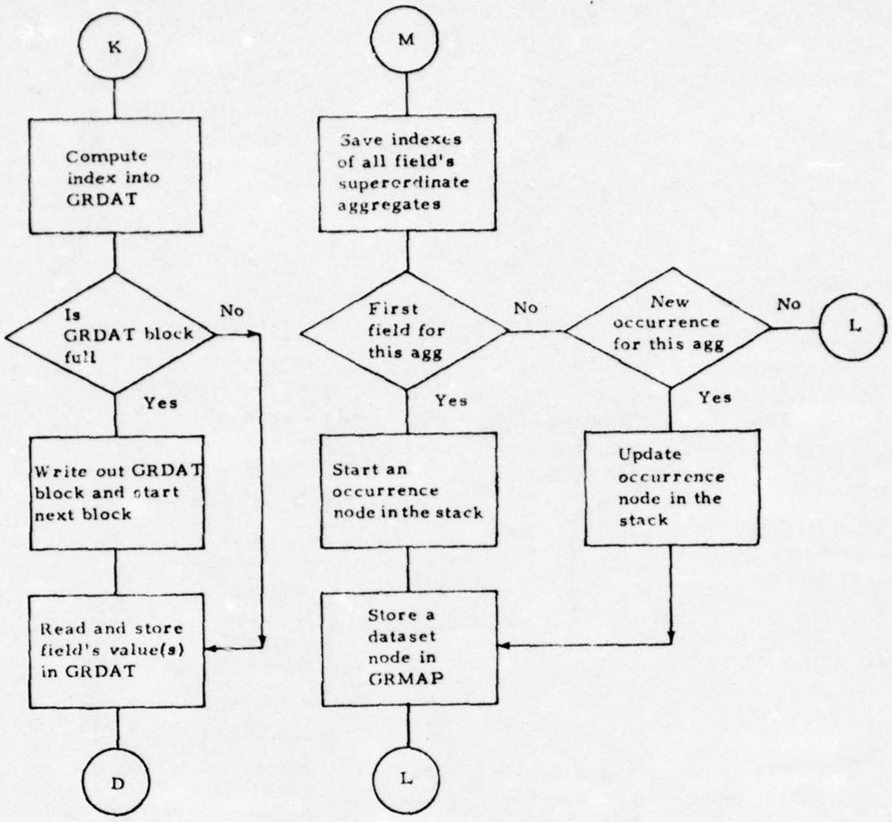

K
Compute index into GRDAT
Is GRDAT block full
No
Yes
Write out GRDAT block and start next block
Read and store field's value(s) in GRDAT
D
M
Save indexes of all field's superordinate aggregates
First field for this agg
No
Yes
New occurrence for this agg
No
Yes
L
Start an occurrence node in the stack
Update occurrence node in the stack
Store a dataset node in GRMAP
L

Figure 43. TSOT Process Data Flow (Sheet 4 of 4).

MAJOR FUNCTION DESCRIPTIONS –

TODATA (Process Character Position Dependent Data) – The first response string for a record is already in TOBUFF. TODATA reads the rest of the response strings needed for a complete record. For each field in the file, TODATA does the following. It picks up the initial character position and size for the field. If there are no data for the field, TODATA goes on to the next field. Otherwise, space in the record data (GRDAT) is allocated and this field's value(s) is stored in GRDAT. A pointer and occurrence count is stored in the record map (GRMAP) for each multivalued field. If a block of record data is filled, it is written out and a new block started. After all fields have been processed, TODATA writes out the record map header (GMPHD), GRMAP, and the final block of GRDAT. TODATA then reads response strings and examines each string of characters until it recognizes either "end of data" and returns or recognizes the start of another record which is processed as in the foregoing.

TOFNAM (Process Field Name Recognition Data) – TOFNAM searches for a field name, reading response strings as needed. If it recognizes "end of data," it stores any occurrence nodes which have been saved in a stack into GRMAP, then writes out the record map header (GMPHD), record map (GRMAP), and final record data block (GRDAT) and returns. If TOFNAM has found a field name, it finds a match in the transformation field data (GTFDS) and saves the field's index and size. TOFNAM determines whether this field starts a new record and, if so, stores occurrence nodes in GRMAP and writes out the files as in the preceding. If the field has no data value, TOFNAM proceeds to the next field. If the field is not in the basic data set, TOFNAM saves the aggregate indexes of the field's parent aggregate and its superordinate aggregates, if any. If this is the *first field* encountered for its aggregate, an occurrence node is started in a stack and a dataset node is stored in GRMAP. If this field starts a new occurrence for its aggregate, the existing occurrence node in the stack is updated and a dataset node is stored in GRMAP. Pointers to the dataset nodes are stored in the occurrence node in the stack with the occurrence count. For all fields which have a data value, TOFNAM calculates an index into GRDAT. If the current GRDAT block is full, it is written out and a new block is started. TOFNAM reads the field's value(s) and stores the data in GRDAT. A pointer and occurrence count are stored in GRMAP for each multivalued and variable-length field. After processing this field, TOFNAM goes on to the next field.

DISPLAY

The DISPLAY process displays selected target system data in a default output format.

GLOBAL DATA USAGE – DISPLAY uses the following global data:

GMPHD – internal record map header.
GRMAP – internal record map.
GRDAT – internal record data.
GTREE – parse tree.
GTDAT – parse tree data.
GFNAM – field names.
GFDES – field descriptions.
GANAM – aggregate names.
GADES – aggregate descriptions.
GFILN – file names.

LOCAL DATA USAGE – DISPLAY uses DILINE – integer array in which each print line is constructed.

GENERAL PROCESS FLOW – DISPLAY is activated whenever a user has input a DISPLAY statement and the corresponding UDL internal records containing the desired target system data are ready for processing. DISPLAY first reads the parse tree. If the user has requested that the output be directed to the high speed printer (HSP) or to a UNIX file by specifying REMOTE on his DISPLAY statement, DISPLAY modifies the output file specification to reflect this request. DISPLAY next reads the file descriptive global data which will be needed to correctly format the data to be displayed. These global data include field names and descriptions (GFNAM, GFDES), aggregate names and descriptions (GANAM, GADES), and file name (GFILN).

For each internal record in the list (implicit or explicit) to be processed, DISPLAY utilizes the following logic. A record header is output specifying the record number. If no display list has been specified in the DISPLAY statement, function DIALLOT is called to output all data in the record. If a display list has been specified, each element is processed in order from the list. A character constant is picked up from the parse tree data (GTDAT) and displayed just as it was input. An aggregate name or an aggregate name qualified by the TREE specification is processed by function DITREE. A field name with no subscripts is processed by function DITREE if it is in an aggregate's data set, otherwise it is processed by function DIFDOUT. A field name with subscripts is processed by function DITREE. After all internal records have been displayed, DISPLAY exits and control returns to the calling process. See figure 44 for more information.

MAJOR FUNCTION DESCRIPTIONS –

DITREE (Display Hierarchical Tree Structure) – DITREE has two input parameters: an aggregate index and a flag which specifies whether the aggregate's tree structure is to be output, only the aggregate is to be output, or only a specified field in the aggregate is to be output.

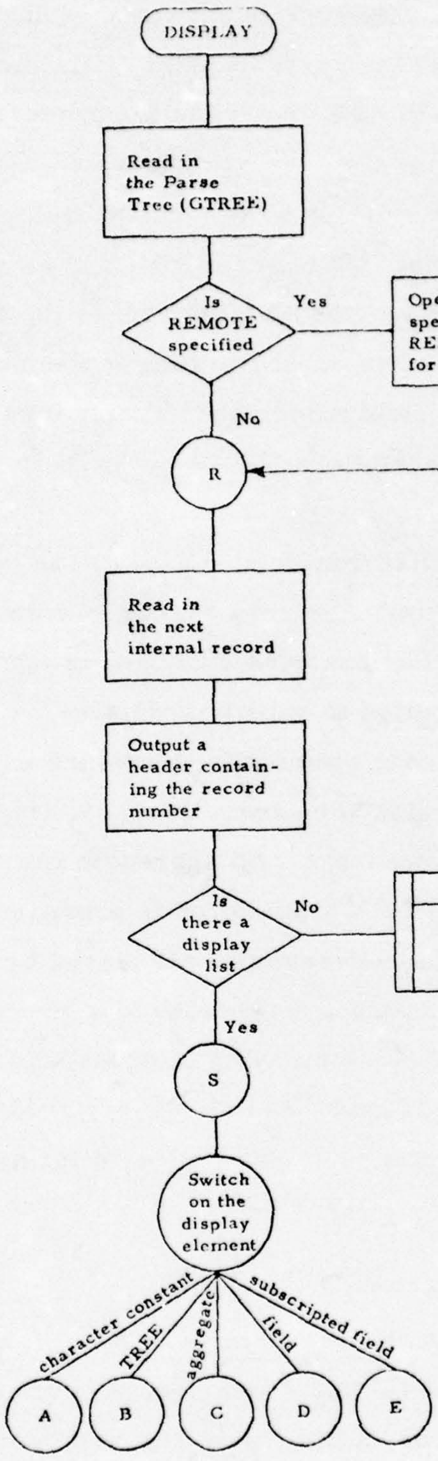

DISPLAY
Read in the Parse Tree (GTREE)
Is REMOTE specified
Yes
Open the specified REMOTE file for output
No
R
Read in the next internal record
Output a header containing the record number
Is there a display list
No
DIALLOT
Output all data for this record
Yes
S
V
Switch on the display element
character constant
TREE
aggregate
field
subscripted field
A
B
C
D
E

Figure 44. DISPLAY Process Data Flow (Sheet 1 of 8).

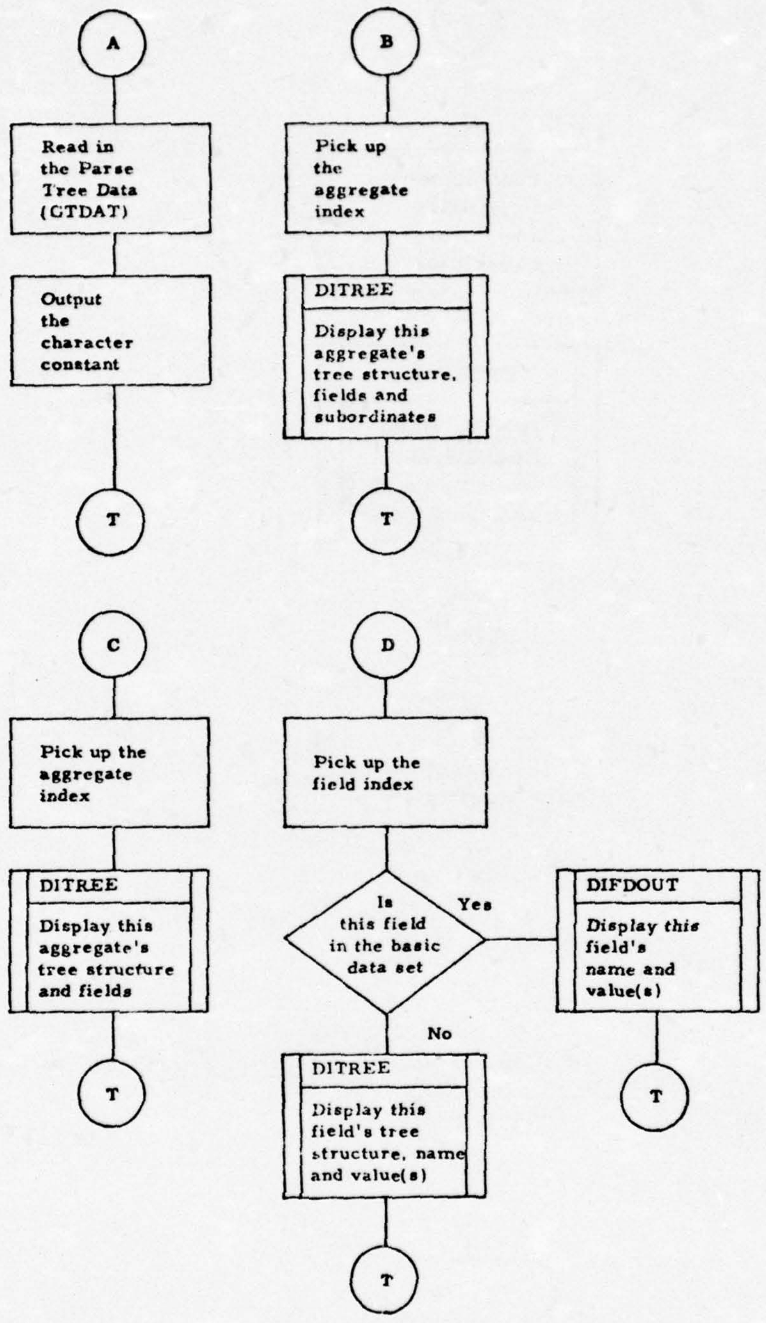

A
Read in the Parse Tree Data (GTDAT)
Output the character constant
T
B
Pick up the aggregate index
DITREE
Display this aggregate's tree structure, fields and subordinates
T
C
Pick up the aggregate index
DITREE
Display this aggregate's tree structure and fields
T
D
Pick up the field index
Is this field in the basic data set
Yes
No
DIFDOUT
Display this field's name and value(s)
T
DITREE
Display this field's tree structure, name and value(s)
T

Figure 44. DISPLAY Process Data Flow (Sheet 2 of 8).

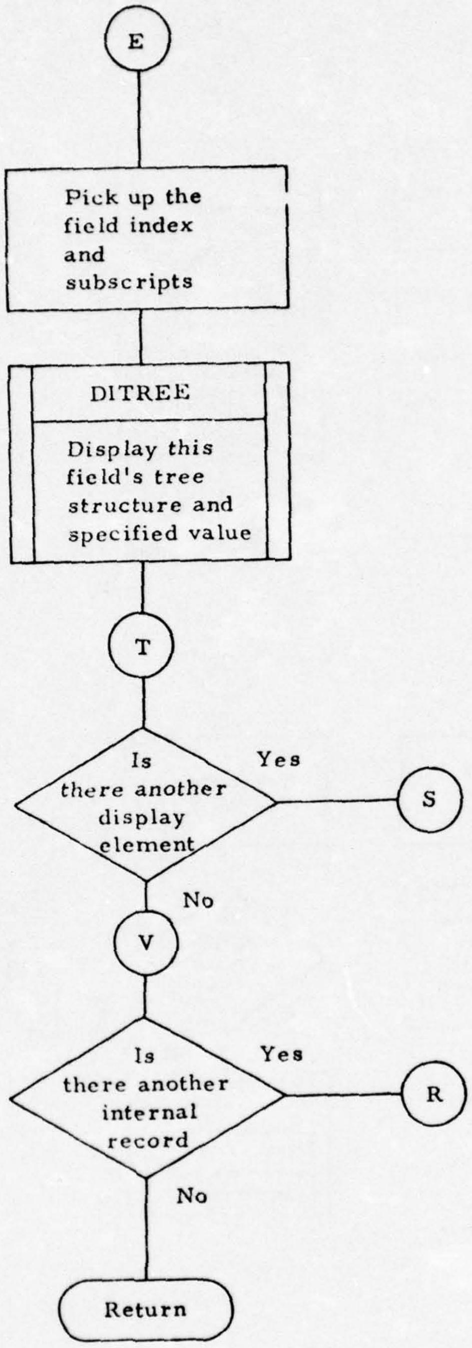

E
Pick up the field index and subscripts
DITREE
Display this field's tree structure and specified value
T
Is there another display element
Yes
S
No
V
Is there another internal record
Yes
R
No
Return

Figure 44. DISPLAY Process Data Flow (Sheet 3 of 8).

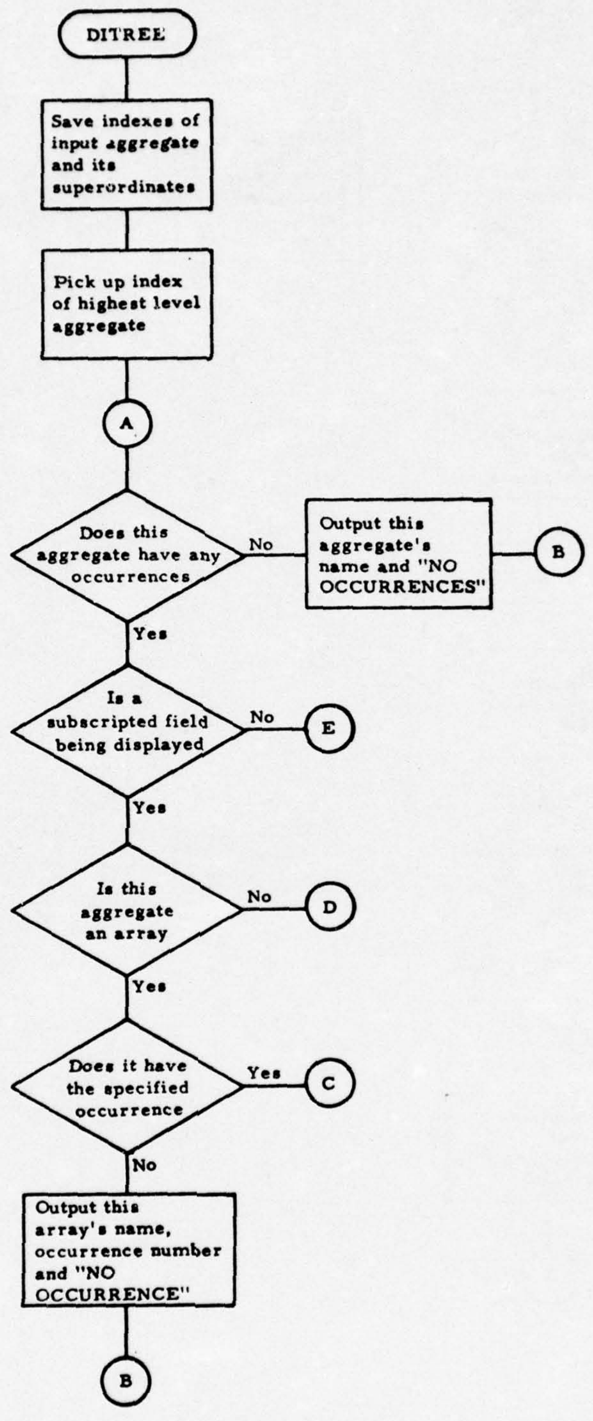

DITREE
Save indexes of input aggregate and its superordinates
Pick up index of highest level aggregate
A
Does this aggregate have any occurrences
No
Output this aggregate's name and "NO OCCURRENCES"
B
Yes
Is a subscripted field being displayed
No
E
Yes
Is this aggregate an array
No
D
Yes
Does it have the specified occurrence
Yes
C
No
Output this array's name, occurrence number and "NO OCCURRENCE"
B

Figure 44. DISPLAY Process Data Flow (Sheet 4 of 8).

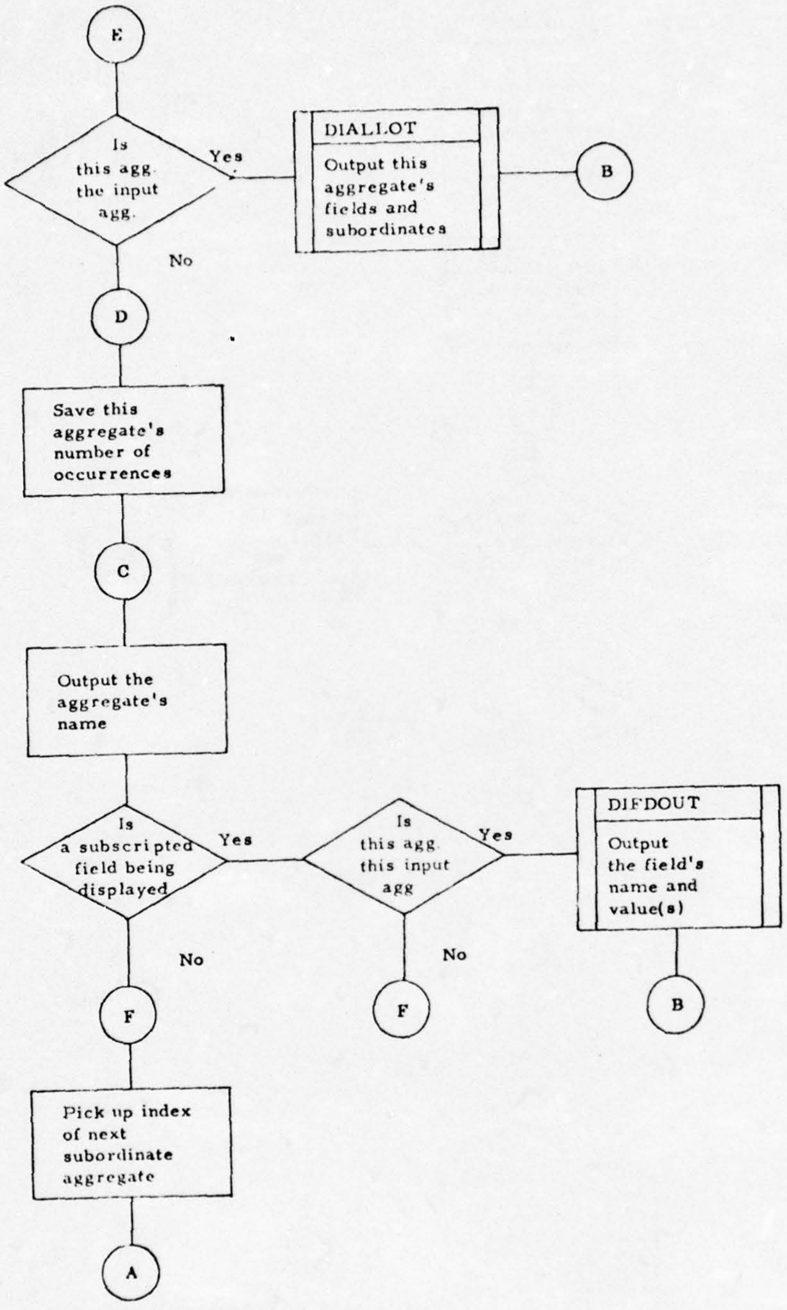

E
Is this agg. the input agg.
Yes
No
DIALLOT
Output this aggregate's fields and subordinates
B
D
Save this aggregate's number of occurrences
C
Output the aggregate's name
Is a subscripted field being displayed
Yes
No
Is this agg. this input agg
Yes
No
DIFDOUT
Output the field's name and value(s)
B
F
F
Pick up index of next subordinate aggregate
A

Figure 44. DISPLAY Process Data Flow (Sheet 5 of 8).

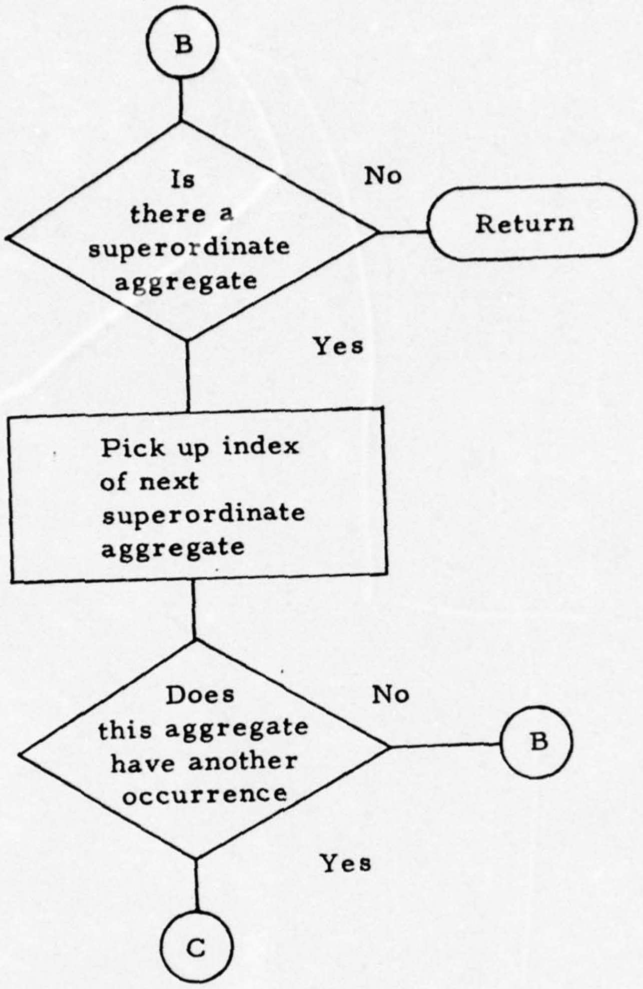

B
Is there a superordinate aggregate
No
Return
Yes
Pick up index of next superordinate aggregate
Does this aggregate have another occurrence
No
B
Yes
C

Figure 44. DISPLAY Process Data Flow (Sheet 6 of 8).

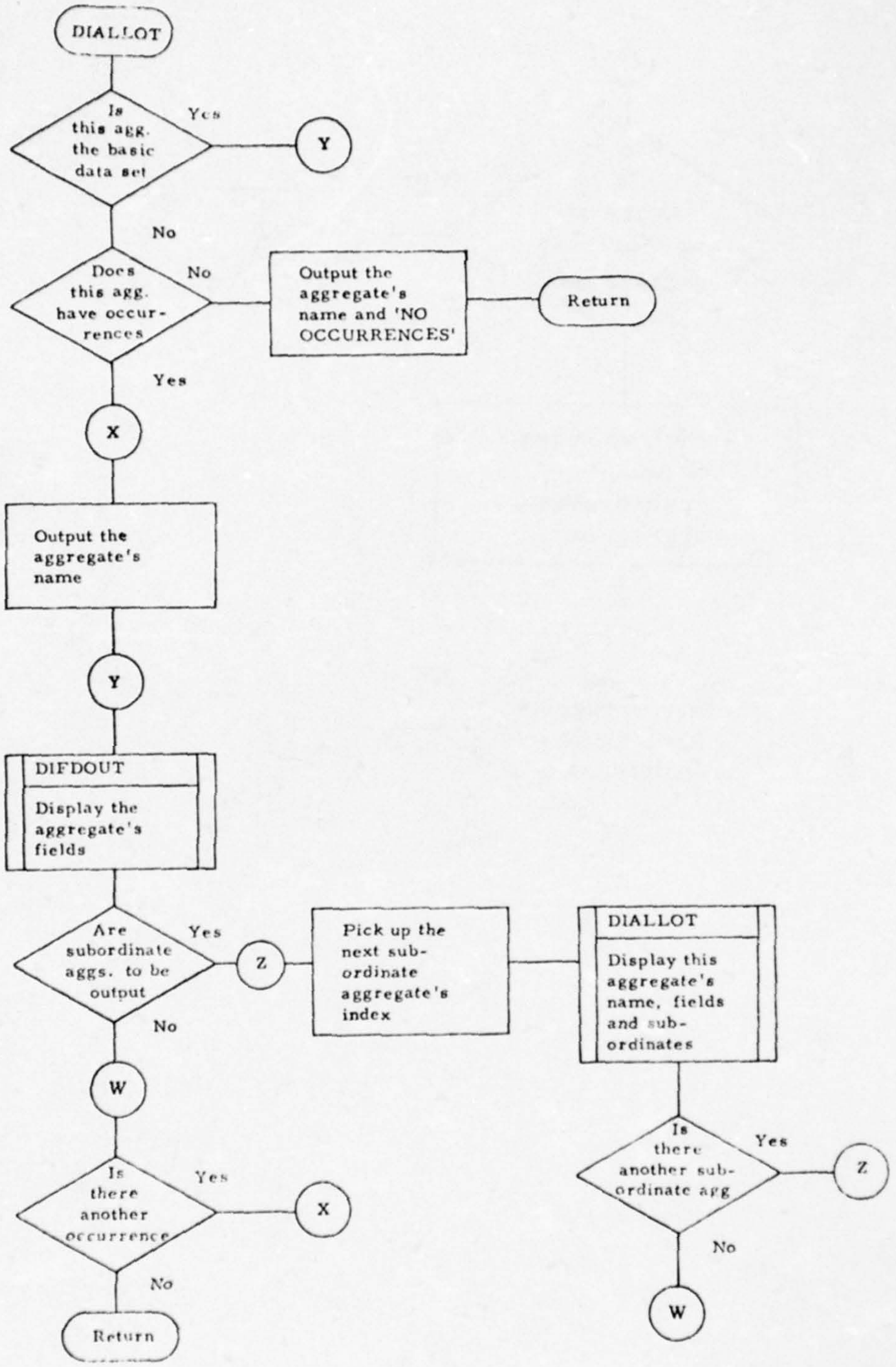

DIALLOT
Is this agg. the basic data set
Yes
Y
No
Does this agg. have occurrences
No
Output the aggregate's name and 'NO OCCURRENCES'
Return
Yes
X
Output the aggregate's name
Y
DIFDOUT
Display the aggregate's fields
Are subordinate aggs. to be output
Yes
Z
Pick up the next subordinate aggregate's index
DIALLOT
Display this aggregate's name, fields and subordinates
No
W
Is there another occurrence
Yes
X
No
Return
Is there another subordinate agg
Yes
Z
No
W

Figure 44. DISPLAY Process Data Flow (Sheet 7 of 8).

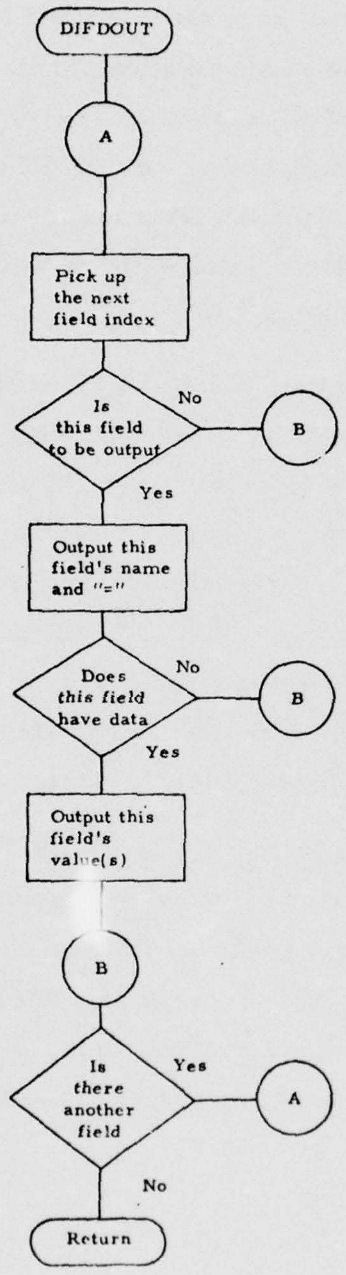

DIFDOUT
A
Pick up the next field index
Is this field to be output
No
B
Yes
Output this field's name and "="
Does this field have data
No
B
Yes
Output this field's value(s)
B
Is there another field
Yes
A
No
Return

Figure 44. DISPLAY Process Data Flow (Sheet 8 of 8).

For each level of aggregate outside of the input aggregate, that aggregate's name is output. If an aggregate has no occurrences at any point in the tree structure, "NO OCCURRENCES" is output for that aggregate. If a subscripted field is being displayed, only the specified array occurrence is output, and, if that occurrence is missing, the array name, occurrence number, and "NO OCCURRENCE" are output. Also, for a subscripted field output, the field data are processed by function DIFDOUT. Otherwise, the input aggregate is processed by function DIALLOT which also receives the input flag value to DITREE. All occurrences of the specified aggregate (or nonsubscripted field) are displayed.

<u>DIALLOT (Display a Data Set)</u> – DIALLOT has three input parameters: an aggregate index; a record map pointer; and a flag which specifies whether all fields and subordinate aggregates for the input aggregate are to be output, all fields and no subordinate aggregates are to be output, or only a specified field is to be output. First, the input aggregate's name is output. If the aggregate has no occurrences, "NO OCCURRENCES is displayed. Otherwise, for each occurrence of the aggregate, the aggregate's fields are output by function DIFDOUT and, if subordinate aggregates are to be output, each one is processed recursively by function DIALLOT.

<u>DIFDOUT (Output an Aggregate's Fields)</u> – DIFDOUT has three input parameters: an aggregate index, a record map pointer, and a flag which specifies whether all fields or a specified field in the input aggregate are to be displayed. For each field to be displayed, the field's name and an equals sign are output. If the field contains data, those data are output either as a single value or as multiple values if the field is multivalued.

ADAPT I GLOBAL FUNCTIONS AND DATA

This section provides descriptions of the global functions and global data now envisioned for ADAPT I. Global functions and data are defined as those functions and data which are commonly utilized by the many processes of ADAPT I. Functions are defined in the literal sense as in the C language, hence are not processes but are actual functions which are compiled with the individual processes requiring their services. Global data are different in this respect since only a single copy of these data exists in the ADAPT I environment. These data are referred to as global because they are usually utilized by many of the ADAPT I processes.

This section is divided into two subsections, one describing global functions and the other providing descriptions of the many global data structures and arrays existing in ADAPT I. Currently only three global functions have been defined for ADAPT I. Fifteen global data structures are currently defined for ADAPT I, and this data set will be expanded with the definition of the TDL dictionaries.

Global Functions

ADAPT I currently requires the use of three global functions:

a. GFERROR – outputs general diagnostic messages.

b. GFLOCK – locks "read-for-update" data files.

c. GFUNLOCK – unlocks "read-for-update" data files.

Following are the descriptions of these functions.

GFERROR FUNCTION –

Input Parameters – GFERROR requires two input parameters:

a. An ADAPT I error message number (integer).

b. An optional parameter pointing to a null-terminated character string.

Output Parameters – None.

Operation – GFERROR provides ADAPT I with a common source for the output of diagnostic-type messages to the user. The first input parameter specifies a canned error message and the second parameter, which is optional, provides a character string which is to be inserted into an appropriate place in the canned message. GFERROR makes the appropriate string substitution in the indicated error message if applicable, and then outputs the message to the user.

GFLOCK FUNCTION –

Input Parameters – GFLOCK requires one input parameter: an integer specifying a global data file.

Output Parameters – None.

Operation – For certain ADAPT I global data, it will be necessary to protect their integrity during update sequences; i.e., for "read-for-update" protection on a given file. That is, only one process may "read-for-update" a given file at any one time. The global function GFLOCK (and its counterpart, GFUNLOCK) provides this protection within the ADAPT I environment.

GFLOCK uses the input parameter to ascertain which global data file is to be "locked out" during a "read-for-update" sequence. This integer is mapped onto a set of lock-files, named lock1, lock2, . . . , lockn, where n equals the integer passed as input to GFLOCK.

Each lock-file, locki, will correspond to a particular ADAPT I global data file. Currently, only three global data files are "read-for-update" sensitive: the user description file, GUSER; the transaction description file, GTDES; and the user list description file, GLIST. These global data structures have been assigned the following lock-files:

GUSER . . . lock1 (input of 1).

GTDES . . . lock2 (input of 2).

GLIST . . . lock3 (input of 3).

GFLOCK attempts to create the designated lock-file with a "read-only" mode. Of course, if this file already exists, it will cause the create call to return with an error. GFERROR "goes to sleep" for a short while and then attempts the create again. Eventually, the create will be successful and GFLOCK will return to the calling function. At this point, the corresponding global data file is "locked out" from other "read-for-update" attempts. Of course, this assumes the cooperation of other processes to initiate GFLOCK prior to attempting to "read-for-update" a file. The file can now be read in, updated, and then written back out. At this point, it is mandatory that GFUNLOCK be called to unlock the global data file.

GFUNLOCK FUNCTION –

Input Parameters – GFUNLOCK requires one input parameter: an integer specifying a global data file.

Output Parameters – None.

Operation – GFUNLOCK is a counterpart to GFLOCK and must be called when a given global file has been updated and written out. Basically, GFUNLOCK removes the associated lock-file thus allowing other processes to create it, hence gaining control of that file. Of course, failure to call GFUNLOCK after successfully locking out the file with GFLOCK would bring the ADAPT I system to a halt.

Global Data

Currently, 15 global data structures have been defined for ADAPT I. These are as follows:

a. GADES – aggregate description file.

b. GANAM – aggregate name file.

c. GFDES – field description file.

d. GFNAM – field name file.

e. GFILD – file description file.

f. GFILN – file name file.

g. GRMAP – record map file.

h. GMPHD – record map header.

i. GRDAT – record data file.

j. GTREE – parse tree file.

k. GTDAT – parse tree data file.

l. GUSER – user description file.

m. GTDES – transaction description file.

n. GLIST – user list description file.

o. GLABL – user statement label description file.

Each global data file is described in detail. All fields comprising a data structure are described. Where applicable (when the global data file is a C structure), an item description is provided illustrating the different fields defined for that global structure.

GADES (AGGREGATE DESCRIPTIONS) – This structure contains descriptions of the data sets (or aggregates) of a specified file. Each item of the structure describes a data set and defines its relationships with other data sets. Item zero contains a description of the basic data set for the file. GADES is built by the Data Definition Language (DDL) process and is used by most of the other processes. GADES is indexed by an aggregate ID

(AGID) which is determined for a given name by an item-by-item search of the aggregate names (GANAM). GANAM is parallel to GADES and, therefore, is indexed by an AGID also. Figure 45 is an item description.

Field Descriptions –

GSIZE	–	size of this aggregate's data; i. e., the number of characters of data for all single-valued fields in this data set.
GPRNT	–	pointer to the aggregate which is superordinate to this aggregate.
GATYP	–	Aggregate type: 0 = basic data set. 1 = repeating group. 2 = array.
GANUM	–	number of aggregates which are directly subordinate to this aggregate; zero if none.
GAPTR	–	pointer to the first aggregate in a linked list of aggregates which are directly subordinate to this aggregate; zero if none.

Word 0	GSIZE	
Word 1	GPRNT	GATYP
Word 2	GANUM	GAPTR
Word 3	GLINK	GMPTR
Word 4	GFNUM	
Word 5	GFPTR	
Word 6	GOCCS	

Figure 45. GADES Item Description.

GLINK	–	pointer to the next aggregate in this aggregate's linked list of sibling aggregates. If GLINK is zero, this aggregate is the last in the list.
GMPTR	–	pointer into an internal record map (GRMAP). GMPTR points into the dataset node of the parent of this aggregate. GMPTR is relative to the start of the dataset node. Refer to GRMAP for further explanation.
GFNUM	–	number of fields which are in this aggregate.
GFPTR	–	pointer to the first field in a sequential list of fields in this aggregate. GFPTR is an index to GFNAM and GFDES.
GOCCS	–	number of occurrences of an array. GOCCS is applicable for GATYP = 2 only.

For the basic data set (index zero), the following fields are not applicable: GPRNT, GLINK, GMPTR, and GOCCS.

GANAM (AGGREGATE NAMES) – This array contains the names of repeating groups and/or arrays defined for a specified file. Each item contains an eight-character name. Item zero is not used. GANAM is built by the Data Definition Language (DDL) process and is used by several other processes. GANAM is indexed by an aggregate ID (AGID) which can be determined for a given name by an item-by-item search. The aggregate descriptions (GADES) is parallel to GANAM and, therefore, is indexed by an AGID also. Figure 46 is a pictorial representation of GANAM.

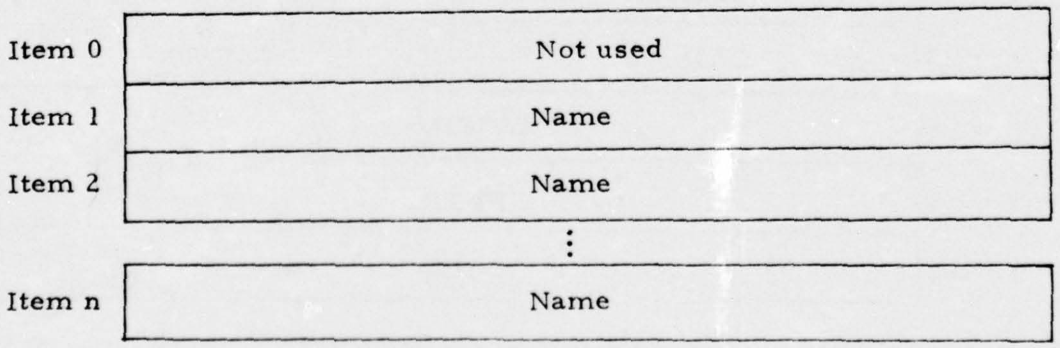


Figure 46. GANAM Pictorial Representation.

<u>GFDES (FIELD DESCRIPTIONS)</u> – This structure contains descriptions of the fields in a specified file. Each item of the structure describes a field. Item zero is not used. GFDES is built by the Data Definition Language (DDL) process and is used by most of the other processes. GFDES is indexed by a field ID (FID) which is determined for a given name by an item-by-item search of the field names (GFNAM). GFNAM is parallel to GFDES and, therefore, is indexed by an FID also. Figure 47 is an item description.

<u>Field Descriptions</u> –

GSIZE	–	maximum size (number of characters) which a value of this field can be. Not applicable for GFTYP = 2.
GPRNT	–	pointer to the aggregate under which this field is defined. GPRNT is an index to GANAM and GADES; i.e., an AGID.
GFTYP	–	Field type: 0 = single-valued. 1 = multivalued. 2 = single-valued variable length.
GDTYP	–	data type of field's value(s): 0 = numeric. 1 = character. 2 = geographic.

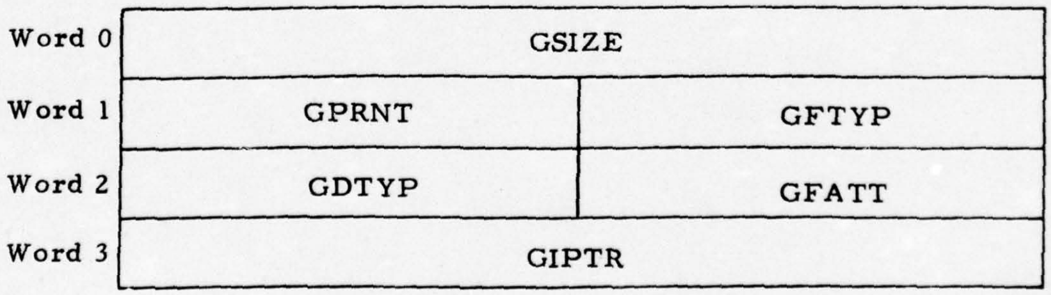
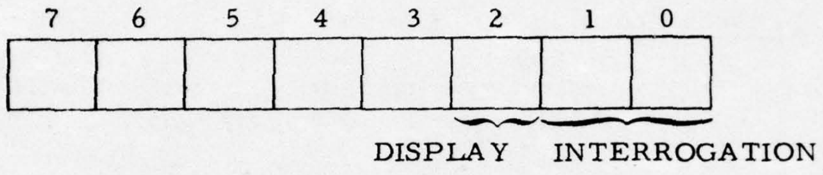


Figure 47. GFDES Item Description.

GFATT	–	field attributes. These data are a bit string which defines the various attributes of the field. See figure 48.
GIPTR	–	pointer into an internal record map (GRMAP) or into an internal record data block (GRDAT). For GFTYP = 1 or 2, GIPTR points into the data set node of the field's parent aggregate. GIPTR is then relative to the start of the data set node. For GFTYP = 0, GIPTR is a pointer to GRDAT to this field's value relative to the start of the parent aggregate's data. Refer to GRMAP for further explanation.



INTERROGATION:

Bits 1	0		
0	0	=	not used.
0	1	=	KEY.
1	0	=	NKEY.
1	1	=	DEP.

DISPLAY:

Bit 2		
0	=	VIS.
1	=	INV.

Figure 48. Field Attribute-Bit Assignments.

GFNAM (FIELD NAMES) – This array contains the names of fields in a specified file. Each item contains an eight-character name. Item zero is not used. GFNAM is built by the Data Definition Language (DDL) process and is used by several other processes. GFNAM is indexed by a field ID (FID) which can be determined for a given name by an item-by-item search. The field descriptions (GFDES) is parallel to GFNAM and, therefore, is indexed by an FID also. Figure 49 is a pictorial representation of GFNAM.

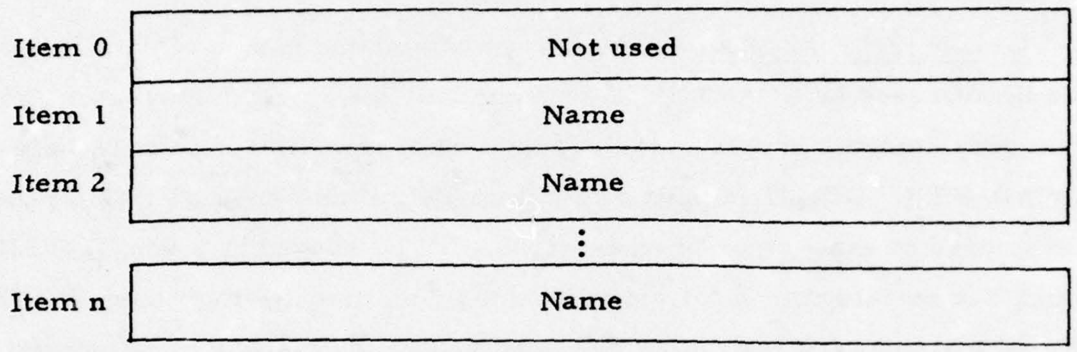


Figure 49. GFNAM Pictorial Representation.

GFILD (FILE DESCRIPTIONS) – This structure contains descriptions of the files which can be accessed by ADAPT I. Each item of the structure describes a file. Item zero is not used. GFILD is built by the Data Definition Language (DDL) process and is used by most of the other processes. GFILD is indexed by a file ID (FLID) which is determined for a given name by an item-by-item search of the file names (GFILN). GFILN is parallel to GFILD and, therefore, is indexed by an FLID also. Figure 50 is an item description.

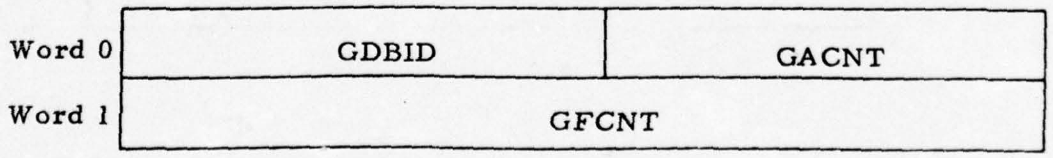


Figure 50. GFILD Item Description.

Field Descriptions –

GDBID	–	data base ID (DBID) of the target system which actually contains the file being referenced.
GACNT	–	number of aggregates defined for this file. GACNT specifies the number of items contained in GANAM and GADES for this file.
GFCNT	–	number of fields defined for this file. GFCNT specifies the number of items contained in GFNAM and GFDES for this file.

GFILN (FILE NAMES) – *This array contains the names of the files which* can be accessed by ADAPT 1. Each item contains an eight-character name. Item zero contains in byte zero the actual number of files currently defined for ADAPT I. GFILN is built by the Data Definition Language (DDL) process and is used by most other processes. GFILN is indexed by a file ID (FLID) which can be determined for a given name by an item-by-item search. The file descriptions (GFILD) is parallel to GFILN and, therefore, is indexed by an FLID also. Figure 51 is a pictorial representation of GFILN.

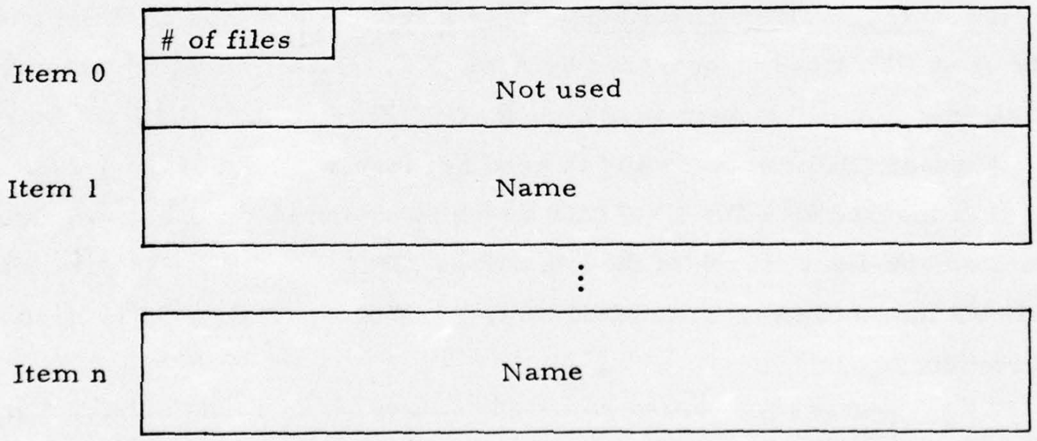


Figure 51. GFILN Pictorial Representation.

GRMAP (INTERNAL RECORD MAP) – This integer array is used to specify the locations of field value(s) for a specific record. Each data record which is sent by a target system to ADAPT is converted into an ADAPT internal record format. Each internal record consists of a record map (GRMAP) and one or more blocks of character data (GRDAT). GRMAP contains absolute pointers into GRDAT, as well as pointers into itself. GRMAP consists of two types of nodes: data set nodes and occurrence nodes. A data set node is related to a specific data set (or aggregate) and contains pointers to GRDAT and to occurrence nodes for subordinate aggregates. An occurrence node is also related to a specific aggregate and contains pointers to data set nodes. The data set node for the basic data set starts in word zero of GRMAP. GRMAP is built by the Target System Output Translator (TSOT) and is used by the DISPLAY process. Descriptions of the two types of nodes in GRMAP follow.

Data Set Node – (See figure 52.)

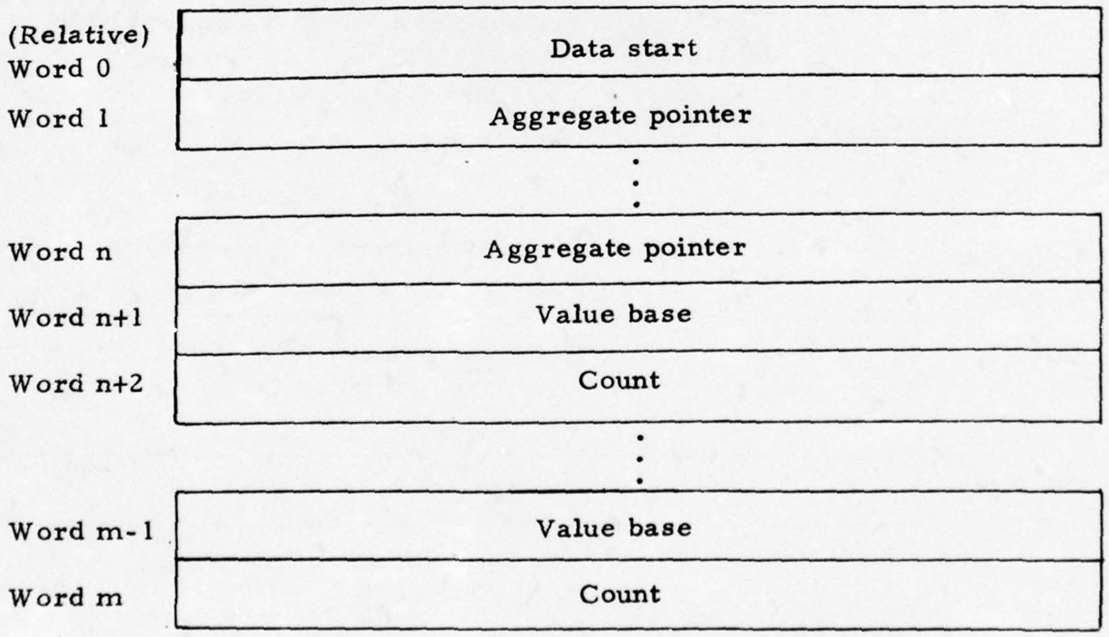


Figure 52. GRMAP Data Set Node Description.

Field Descriptions –

Data start	–	pointer into GRDAT to the start of this aggregate's data values (for single-valued field only). If this aggregate has no single-valued fields or if there are no values for those fields, "data start" will be null.
Aggregate pointer	–	pointer into GRMAP to an occurrence node for a specific aggregate. There is an "aggregate pointer" for each aggregate which is directly subordinate to the aggregate whose data set node this is. This field is pointed to by field GMPTR of the aggregate descriptions (GADES). If the "aggregate pointer" is null, this aggregate has no occurrences at this point in the record.
Value base	–	pointer into GRDAT to the start of a field's data value(s). There is a "value base/count" for each multivalued and single-valued variable length field in the aggregate whose data set node this is. This field is pointed to by field GIPTR of the field descriptions (GFDES). If the field has no data, the "value base" is null.
Count	–	number of occurrences of a multivalued field or number of characters in a single-valued variable length field's data.

Occurrence Node – (See figure 53.)

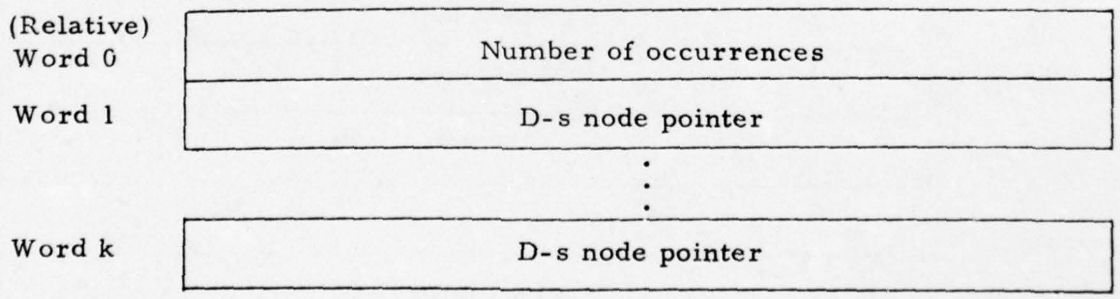


Figure 53. GRMAP Occurrence Node Description.

Field Descriptions –

Number of occurrences	–	number of occurrences of the aggregate whose occurrence node this is.
D-s node pointer	–	pointer into GRMAP to a data set node for this occurrence of the aggregate.

<u>GMPHD (INTERNAL RECORD MAP HEADER)</u> – This integer array contains information about the internal record map which it precedes when written onto a disk file. GMPHD contains size information which is required by ADAPT I when reading a sequence of record maps and record data. Figure 54 is a pictorial representation of GMPHD.

Word 0	File ID
Word 1	Map size
Word 2	Data start
Word 3	Number blocks
Word 4	Data size

Figure 54. GMPHD Description.

<u>Field Descriptions</u> –

File ID	–	pointer to the file to which the following record map (GRMAP) and its associated record data blocks (GRDAT) belong. The file ID (FLID) is an index to GFILN and GFILD.
Map size	–	number of words actually written out for the following record map.

Data start	–	word number within the record data disk file at which this record's data start.
Number blocks	–	number of blocks of data actually written out for this record. Maximum block size is 512 bytes.
Data size	–	number of words actually written out for the final block of record data.

<u>GRDAT (INTERNAL RECORD DATA)</u> – This character array is used to hold data values for a specific record. Index zero is not used. GRDAT is indexed by pointers stored in the record map (GRMAP) in combination with relative pointers stored in the field descriptions (GFDES). Single-valued fields which have no values will have ASCII blanks stored for them.

<u>GTREE (PARSE TREE)</u> – This integer array contains the parse tree for a specific statement. Each UDL statement/command consists of terminal and nonterminal elements. In the parse tree, the pertinent elements in a statement correspond to terminal and nonterminal nodes. A nonterminal node contains a token type describing the nonterminal element and, perhaps, pointers to other nodes. A terminal node contains a token type describing the terminal element and a pointer into the parse tree data (GTDAT). A two-word header starts in word zero of GTREE. GTREE is built by the lexical analyzer portion of either the Executive (EXEC) process or the Data Definition Language (DDL) process. GTREE is used by most processes in ADAPT. Descriptions of the header and nodes follow.

<u>Header</u> – (See figure 55.)

Word 0	Initial pointer
Word 1	Label flag

Figure 55. GTREE Header.

Field Descriptions –

Initial pointer	–	pointer into GTREE to the primary nonterminal node for this statement.
Label flag	–	flag to specify whether this statement has a label. 0 = no; 1 = yes. If yes, the label is stored in the first eight bytes of the parse tree data (GTDAT).

Nonterminal Node (Type 1) – (See figure 56.)

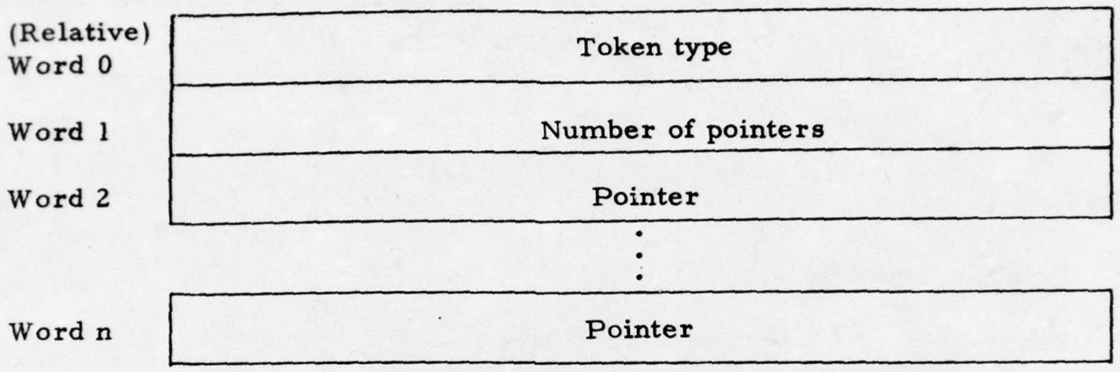


Figure 56. GTREE Nonterminal Node (Type 1).

Nonterminal Node (Type 2) – (See figure 57.)

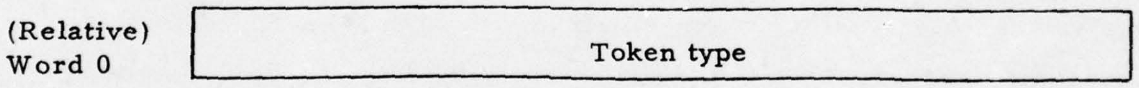


Figure 57. GTREE Nonterminal Node (Type 2).

Field Descriptions –

Token type	–	nonterminal token type (100 – 599). For keywords with no arguments (e.g., ECHO), a type 2 node is stored in the parse tree.

Number of pointers –	number of pointers following this word (can be zero).
Pointer –	pointer into GTREE to another node.

Terminal Node – (See figure 58.)

(Relative)	
Word 0	Token type
Word 1	Data pointer
Word 2	Number of characters

Figure 58. GTREE Terminal Node.

Field Descriptions –

Token type –	terminal token type (1 – 99).
Data pointer –	pointer into the parse tree data (GTDAT) to the data associated with this parse tree node.
Number of characters –	number of characters of data stored in GTDAT for this parse tree node.

GTDAT (PARSE TREE DATA) – This character array is used to hold data values for a specific statement. Index zero is not used. GTDAT is indexed by pointers stored in the parse tree (GTREE). GTDAT is built by the lexical analyzer portion of either the Executive (EXEC) process or the Data Definition Language (DDL) process. GTDAT is used by most processes in ADAPT I.

GUSER (USER DESCRIPTION) – This structure contains information pertinent to all users that utilize ADAPT I. The establishment of an ADAPT I user is accomplished by the ADAPT I Sysgen Program. During this process, the userid is entered and an ADAPT I superuser may be established (there can be only one superuser). A user must be established in GUSER prior to using ADAPT I. Internally, users are recognized by an index (UID) into this structure. A UID of zero (item-0 of GUSER) implies the ADAPT I superuser. Figure 59 is an item description.

Field Descriptions –

GUSID	–	user-identifier, 1 – 10 characters. Must be unique within GUSER.
GSTAT	–	contains user status: 0 = user is *not* logged-on to ADAPT I; 1 = user is logged-on to ADAPT I.
GNLTN	–	contains the next logical transaction number (LTN) for this user. LTNs are applicable for batch transactions only. A batch transaction is uniquely identified for a user by the concatenation of the user's UID with the appropriate LTN.
GCLTI	–	contains the current index into the transaction description structure (GTDES) for this user.

Words 0 – 4	GUSID	
Word 5	GSTAT	CNLTN
Word 6	GCLTI	Not used

Figure 59. GUSER Item Description.

<u>GTDES (TRANSACTION DESCRIPTIONS)</u> – This structure contains information pertinent to ADAPT I logical transactions. Logical transactions are applicable for batch queries only. Entries into GTDES are made by the Target System Input Generator:Batch (TSIGB) process and are removed by the Network Response (NETRES) process. Entries in GTDES are identifiable by a five-character external transaction number constructed by the concatenation of a two-digit user identifier (UID), and a three-digit logical transaction number (LTN). GTDES entries remain in GTDES until a target system has responded and a user has perused (via the VIEW command) the associated batch output. Figure 60 is an item description.

<u>Field Descriptions</u> –

GELTN	–	external logical transaction number, five characters. Is unique in GTDES. User identifier and LTN are contained in this field.
GLSID	–	record list index. Points to the appropriate GLIST item which will represent (either as an explicit or implicit list) the transaction output.
GTMOD	–	Transaction mode: 1 = active, waiting for the target system response (i. e., an ANSR message, an ABRT (FAULT), etc.); 2 = completed, waiting for the user to peruse the query results.

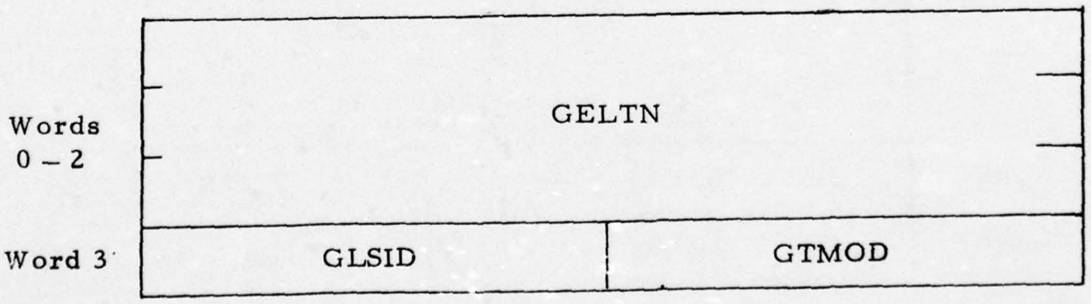


Figure 60. GTDES Item Description.

Item-0 of GTDES specifies the number of logical transaction entries currently residing in GTDES.

GLIST (LIST DESCRIPTIONS) – This structure contains information pertinent to UDL record lists. Lists can originate either as an explicit list, due to a user entering a SAVE command, or as an implicit list generated internally by ADAPT I, caused by a user entering a DISPLAY statement. The explicit list will exist indefinitely until the user deletes it. An implicit list exists only for the duration of time required to display its records. All list-names are unique for a given user. Internal list-names will be generated uniquely in order to avoid conflict with user-specified names. Entries into GLIST are made by the Target System Input Generator processes (TSIGI and TSIGB). Figure 61 is an item description.

Field Descriptions –

GLSTN	–	list-name, 1 – 8 characters. Must be unique within GLIST.
GLSFD	–	contains file-identifier (FLID, refer to global data structures GFILN and GFLID) of file from which list-records originated.
GLSTY	–	list type: 0 = implicit list; 1 = explicit list.

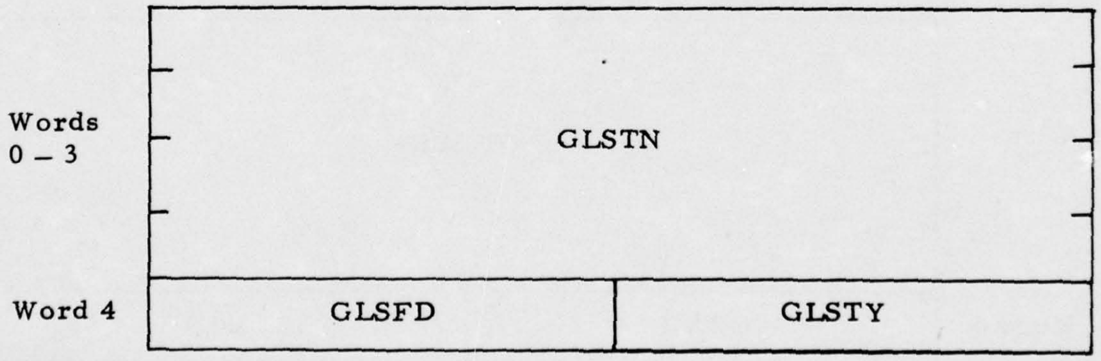


Figure 61. GLIST Item Description.

Item-0 of GLIST is used to contain the number of lists currently residing in GLIST.

GLABL (LABEL DESCRIPTIONS) – This structure contains information pertinent to user statement labels that are defined during any given logon-sequence. Labels are applicable on FIND statements only (for ADAPT I) and exist only for the duration of the user's logon-sequence. That is, once a user logs-off ADAPT I, all FIND statement labels are deleted. Also a user may only have a specified number of labels active at any one time, the oldest label being deleted in order to accommodate the new label. Labels have significance for dependent FIND statement references and for DISPLAY and SAVE record source specifications. Labels are inserted by the Target System Input Generator (TSIGI and TSIGB) processes and are deleted when the user initiates an ADAPT I LOGOFF command. Figure 62 is an item description.

Field Descriptions –

GLBEL	–	label, 1 – 8 characters. Must be unique within GLABL.
GFLID	–	contains the file-identifier (FLID, refer to global data structures GFILN and GFILD) of the file referenced by the FIND statement (either explicitly via "IN file-name" or implicitly via "SOURCE(tag)").

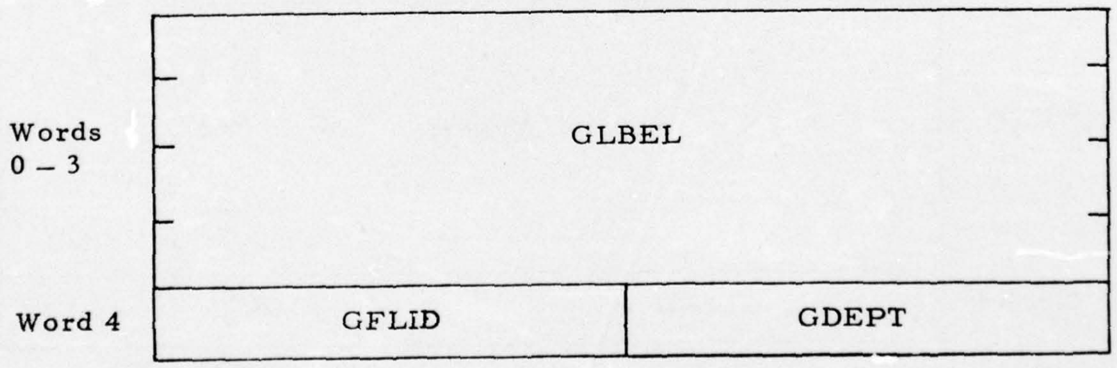


Figure 62. GLABL Item Description.

GDEPT	–	indicator for FIND statement type: 0 = label is associated with an independent FIND statement; ≠0 = label is associated with a dependent FIND statement. For this case, GDEPT contains an index into GLABL to the referenced FIND statement.

Item-0 of GLABL is used to contain the number of labels currently residing in GLABL.

APPENDIX A

ADAPT I YACC SPECIFICATIONS

YACC (Yet Another Compiler-Compiler) is utilized for "front-end" development in ADAPT I. This appendix presents the UDL/DDL grammars as YACC specifications. YACC accepts left associative LR(1) grammars as input and will resolve any syntactic ambiguities if they occur. The output from YACC is a parser program and an LR(1) parse table representing the grammar. Not shown in the YACC specifications presented herein are the user-supplied lexical analyzer and parse tree generators.

ADAPT I will utilize YACC for three separate "front-ends": 1) a "front-end" for parsing UDL statements and commands, 2) a "front-end" for DDL statements, and 3) a "front-end" for Transformation Definition Language (TDL) statements. TDL is still in the design stage and, therefore, is not represented in this appendix. Subsequent pages provide the YACC specifications for UDL and DDL.

UDL YACC Specification

```
%token    '\n' 0
%token    NAME 1
%token    CHARCONS 30 NUMCONS 31 GEOCONS 32
%token    OR 100 AND 101 XOR 102 NOR 103 NOT 104
%token    EQ 140 GT 141 LT 142 GE 143 LE 144 WRG 145 ORG 146
%token    INSIDE 150 OUTSIDE 151 ALONG 152
%token    IN 300 SOURCE 301 CIRCLE 302 POLY 303 ROUTE 304
%token    REMOTE 305 HSP 306 TREE 307 LIST 308 ECHO 309
%token    VALIDATE 310 ACTION 311 SNAME 312 GEOLIST 313
%token    DISPLIST 314 OPENLIST 315 FNAMLIST 316 ACCESS 317
%token    OPITM 318 SCOPE 319
%token    DATABASE 412
%token    FIND 500 DISPLAY 501
%token    LOGOFF 520 OPEN 521 CLOSE 522 EXECUTE 523 MODE 524
%token    SAVE 525 DELETE 526 VIEW 527 TRANSMIT 528
```

```
%token    SCHEMA 550
%token    '(' 490 ')' 491 ',' 492
%token    LEXERR 600
%%                                            /*  beginning of rules section */
list:                                         /*  list is the start symbol */
                                              /*  empty */
          | list stat '\n'                    /*  statement */
                  = {psfinsh($2);}
          | list error '\n'                   /*  syntax error */
                  = {psfinsh(0) ; } ;
stat:     FIND sclause selcrit                /*  FIND statement */
                  = {$$ = pstree2(FIND, $2, $3);}
          | DISPLAY SOURCE '(' NAME ')'       /*  DISPLAY statement */
                  = {$$ = pstree1(DISPLAY, $4);}
          | DISPLAY SOURCE '(' NAME ')' dlist
                  = {$$ = pstree2(DISPLAY, $4, $6);}
          | DISPLAY SOURCE '(' NAME ')' locspec
                  = {$$ = pstree2(DISPLAY, $4, $6);}
          | DISPLAY SOURCE '(' NAME ')' dlist locspec
                  = {$$ = pstree3(DISPLAY, $4, $6, $7);}
          | LOGOFF                            /*  LOGOFF command */
                  = {$$ = pstreeno(LOGOFF);}
          | OPEN oplist                       /*  OPEN command */
                  = {$$ = pstree1(OPEN, $2);}
          | SAVE SOURCE '(' NAME ')' NAME     /*  SAVE command */
                  = {$$ = pstree2(SAVE, $4, $6);}
          | DELETE dclause                    /*  DELETE command */
                  = {$$ = pstree1(DELETE, $2);}
          | VIEW vclause                      /*  VIEW command */
                  = {$$ = pstree1(VIEW, $2);}
          | CLOSE fnlist                      /*  CLOSE command */
                  = {$$ = pstree1(CLOSE, $2);}
          | CLOSE
                  = {$$ = pstree0(CLOSE);}
          | MODE modespec                     /*  MODE command */
                  = {$$ = pstree1(MODE, $2);}
          | EXECUTE namespec                  /*  EXECUTE command */
                  = {$$ = pstree1(EXECUTE, $2);}
          | EXECUTE namespec echospec
                  = {$$ = pstree2(EXECUTE, $2, $3);}
          | EXECUTE namespec locspec
                  = {$$ = pstree2(EXECUTE, $2, $3);}
          | EXECUTE namespec echospec locspec
                  = {$$ = pstree3(EXECUTE, $2, $3, $4);}
          | SCHEMA NAME DATABASE '(' NAME ')' NAME          /*  SCHEMA statement */
                  = {$$ = pstree3(SCHEMA, $2, $5, $7);}
          | TRANSMIT NAME                                   /*  TRANSMIT command */
                  = {$$ = pstree1(TRANSMIT, $2);} ;
sclause:                                      /*  source clause for FIND statement */
          IN NAME
                  = {$$ = pstree1(IN, $2);}
          | SOURCE '(' NAME ')'
                  = {$$ = pstree1(SOURCE, $3);} ;
selcrit:                                      /*  selection criteria */
          selclause
          | selcrit OR selclause
                  = {$$ = pstree2(OR, $1, $3);} ;
```

```
selclause:
          selphrase
        | selclause AND selphrase
                  = {$$ = pstree2(AND, $1, $3);};
selphrase:
          selfactor
        | selphrase XOR selfactor
                  = {$$ = pstree2(XOR, $1, $3);}
        | selphrase NOR selfactor
                  = {$$ = pstree2(NOR, $1, $3);};
selfactor:
          selprimary
        | NOT selprimary
                  = {$$ = pstree1(NOT, $2);};
selprimary:
          selterm
        | '(' selcrit ')'
                  = {$$ = $2;}
        | NAME '(' selcrit ')'
                  = {$$ = pstree2(SCOPE, $1, $3);};
selterm:
          fterm EQ datacons
                  = {$$ = pstree2(EQ, $1, $3);}
        | fterm GT NUMCONS
                  = {$$ = pstree2(GT, $1, $3);}
        | fterm LT NUMCONS
                  = {$$ = pstree2(LT, $1, $3);}
        | fterm GE NUMCONS
                  = {$$ = pstree2(GE, $1, $3);}
        | fterm LE NUMCONS
                  = {$$ = pstree2(LE, $1, $3);}
        | fterm WRG NUMCONS ',' NUMCONS
                  = {$$ = pstree3(WRG, $1, $3, $5);}
        | fterm ORG NUMCONS ',' NUMCONS
                  = {$$ = pstree3(ORG, $1, $3, $5);}
        | fterm INSIDE geoexp
                  = {$$ = pstree2(INSIDE, $1, $3);}
        | fterm OUTSIDE geoexp
                  = {$$ = pstree2(OUTSIDE, $1, $3);}
        | fterm ALONG geoexp
                  = {$$ = pstree2(ALONG, $1, $3);};
fterm:
          NAME
        | subname;
subname:                                        /* subscripted field name */
          NAME '(' NUMCONS ')'
                  = {$$ = pstree2(SNAME, $1, $3);}
        | NAME '(' NUMCONS ',' NUMCONS ')'
                  = {$$ = pstree3(SNAME, $1, $3, $5);}
        | NAME '(' NUMCONS ',' NUMCONS ',' NUMCONS ')'
                  = {$$ = pstree4(SNAME, $1, $3, $5, $7);};
datacons:                                       /* data constant */
          NUMCONS
        | CHARCONS
        | GEOCONS;
geoexp:                                         /* geographic expression */
          CIRCLE '(' NUMCONS ',' GEOCONS ')'
                  = {$$ = pstree2(CIRCLE, $3, $5);}
          | POLY '(' gelist ')'
                   = {$$ = pstreel(POLY, $3);}
          | ROUTE '(' NUMCONS ',' gelist ')'
                   = {$$ = pstree2(ROUTE, $3, $5);};
                                            /*  geographic constant list */
gelist:
          glist
                   = {$$ = pslstdon(GEOLIST);};

glist:
          GEOCONS
                   = {pslstnew($1);}
          | glist ',' GEOCONS
                   = {pslstnxt($3);};
                                            /*  display list */
dlist:
          dislist
                   = {$$ = pslstdon (DISPLIST);};
dislist:
          dispelem
                   = {pslstnew($1);}
          | dislist ',' dispelem
                   = {pslstnxt($3);};
                                            /*  display element */
dispelem:
          CHARCONS
          | fterm
          | TREE '(' NAME ')'
                   = {$$ = pstreel(TREE, $3);};
                                            /*  output location specification */
locspec:
          REMOTE '(' remarg ')'
                   = {$$ = pstreel(REMOTE, $3);};
                                            /*  REMOTE argument */
remarg:
          CHARCONS
          | HSP
                   = {$$ = pstreeno(HSP);};
                                            /*  open list */
oplist:
          opnlist
                   = {$$ = pslstdon(OPENLIST);};
opnlist:
          opitem
                   = {pslstnew($1);}
          | opnlist ',' opitem
                   = {pslstnxt($3);};
opitem:
          NAME '(' NAME ')'
                   = {$$ = pstree2(OPITM, $1, $3);};
                                            /*  DELETE clause */
dclause:
          LIST '(' NAME ')'
                   = {$$ = pstreel(LIST, $3);}
          | SCHEMA '(' NAME ')'
                   = {$$ = pstreel(SCHEMA, $3);};
                                            /*  VIEW clause */
vclause:
          LIST
                   = {$$ = pstreeno(LIST);}
          | SCHEMA '(' NAME ')'
                   = {$$ = $3;};
fnlist:                                     /*  file name list */
          filelist
                   = {$$ = pslstdon(FNAMLIST);};
```

```
filelist:
          NAME
                    = {pslstnew($1);}
        | filelist ',' NAME
                    = {pslstnxt($3);};
modespec:                                     /* MODE specification */
          VALIDATE
                    = {$$ = pstreeno(VALIDATE);}
        | ACTION
                    = {$$ = pstreeno(ACTION);};
namespec:                                     /* unix-file specification for EXECUTE */
          CHARCONS
        | SCHEMA '(' CHARCONS ')'
                    = {$$ = pstreel(SCHEMA, $3);};
echospec:
          ECHO
                    = {$$ = pstreeno(ECHO);};
```

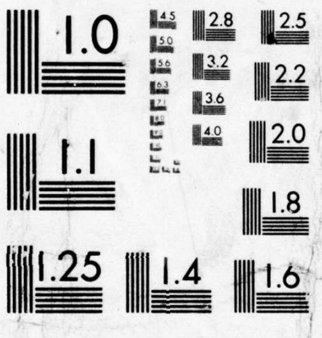
DDL YACC Specification

```
%token    '\n' 0
%token    NAME 1
%token    INTEGER 33
%token    ATT 400 SV 401 MV 402 CHAR 403 NUM 404 GEO 405 KEY 406 NKEY 407
%token    DEP 408 VIS 409 INV 410 V 411 DATABASE 412
%token    SCHEMA 550 ESCHEMA 551 FIELD 552 ARRAY 553
%token    EARRAY 554 RGROUP 555 ERGROUP 556
%token    '(' 490 ')' 491 ',' 492
%token    LEXERR 600
%%                                            /* beginning of rules section */
list:                                         /* list is the start symbol */
                                              /* empty */
        | list stat '\n'                      /* statement */
                    = {psfinish($2);}
        | list error '\n'                     /* syntax error */
                    = {psfinish(0);};
stat:
          SCHEMA NAME DATABASE '(' NAME ')' NAME          /* SCHEMA statement */
                    = {$$ = pstree3(SCHEMA, $2, $5, $7);}
        | ESCHEMA                             /* ESCHEMA statement */
                    = {$$ = pstreeno(ESCHEMA);}
        | FIELD NAME ftyp dtyp ATT '(' iatt ',' datt ')' size     /* FIELD statement */
                    = {$$ = pstree6(FIELD, $2, $3, $4, $7, $9, $11);}
        | ARRAY NAME INTEGER                  /* ARRAY statement */
                    = {$$ = pstree2(ARRAY, $2, $3);}
        | ARRAY NAME INTEGER NAME
                    = {$$ = pstree3(ARRAY, $2, $3, $4);}
        | EARRAY                              /* EARRAY statement */
                    = {$$ = pstreeno(EARRAY);}
        | RGROUP NAME                         /* RGROUP statement */
                    = {$$ = pstreel(RGROUP, $2);}
        | RGROUP NAME NAME
                    = {$$ = pstree2(RGROUP, $2, $3);}
        | ERGROUP                             /* ERGROUP statement */
                    = {$$ = pstreeno(ERGROUP);};
```

AD-A032 006   LOGICON INC SAN DIEGO CALIF   F/G 9/2
ADAPT I PRELIMINARY FUNCTIONAL AND SYSTEM DESIGN SPECIFICATION. (U)
NOV 76 L R ERICKSON, M E SOLEGLAD   N00014-76-C-0899
UNCLASSIFIED   76-C-0899-2   NL

3 OF 3
AD
A032006

END
DATE
FILMED
1 — 77


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

```
ftyp:                                      /*  field type */
        SV
              = {$$ = pstreeno(SV);}
        | MV
              = {$$ = pstreeno(MV);};
dtyp:                                      /*  data type */
        CHAR
              = {$$ = pstreeno(CHAR);}
        | NUM
              = {$$ = pstreeno(NUM);}
        | GEO
              = {$$ = pstreeno(GEO);};
iatt:                                      /*  interrogation attribute */
        KEY
              = {$$ = pstreeno(KEY);}
        | NKEY
              = {$$ = pstreeno(NKEY);}
        | DEP
              = {$$ = pstreeno(DEP);};
datt:                                      /*  display attribute */
        VIS
              = {$$ = pstreeno(VIS);}
        | INV
              = {$$ = pstreeno(INV);};
size:                                      /*  field size */
        INTEGER
        | INV
              = {$$ = pstreeno(V);};
```

APPENDIX B

ADAPT I FILE DICTIONARY AND RECORD MAP UTILIZATION

This appendix provides illustrative material which will better describe how ADAPT I stores and utilizes a target system file's description and internal data records for that file. Figure B-1 shows the logical structure of a fictitious PERSONNL file as it would be visualized by ADAPT I. This file contains fields describing each employee's present status, family situation, and past job experience. Each employee's basic data set (BDS) consists of his first and last names, his title and salary, and, if married, his/her spouse's first name. Subordinate to the basic data set are two repeating groups, one describing children and one describing past job experience. Each occurrence of the CHILDREN repeating group describes a child and consists of the child's first name and hobbies. Each occurrence of the EXPERNCE repeating group describes a previous job and consists of the company's name and the number of years worked. Subordinate to the EXPERNCE repeating group is another repeating group describing projects worked on. Each occurrence of the PROJECTS repeating group describes a project and consists of the project name and the supervisor's name. In order to go from the logical file structure in figure B-1 to the file dictionary data shown in figure B-2, a Data Definition Language (DDL) SCHEMA run must be input to ADAPT I.

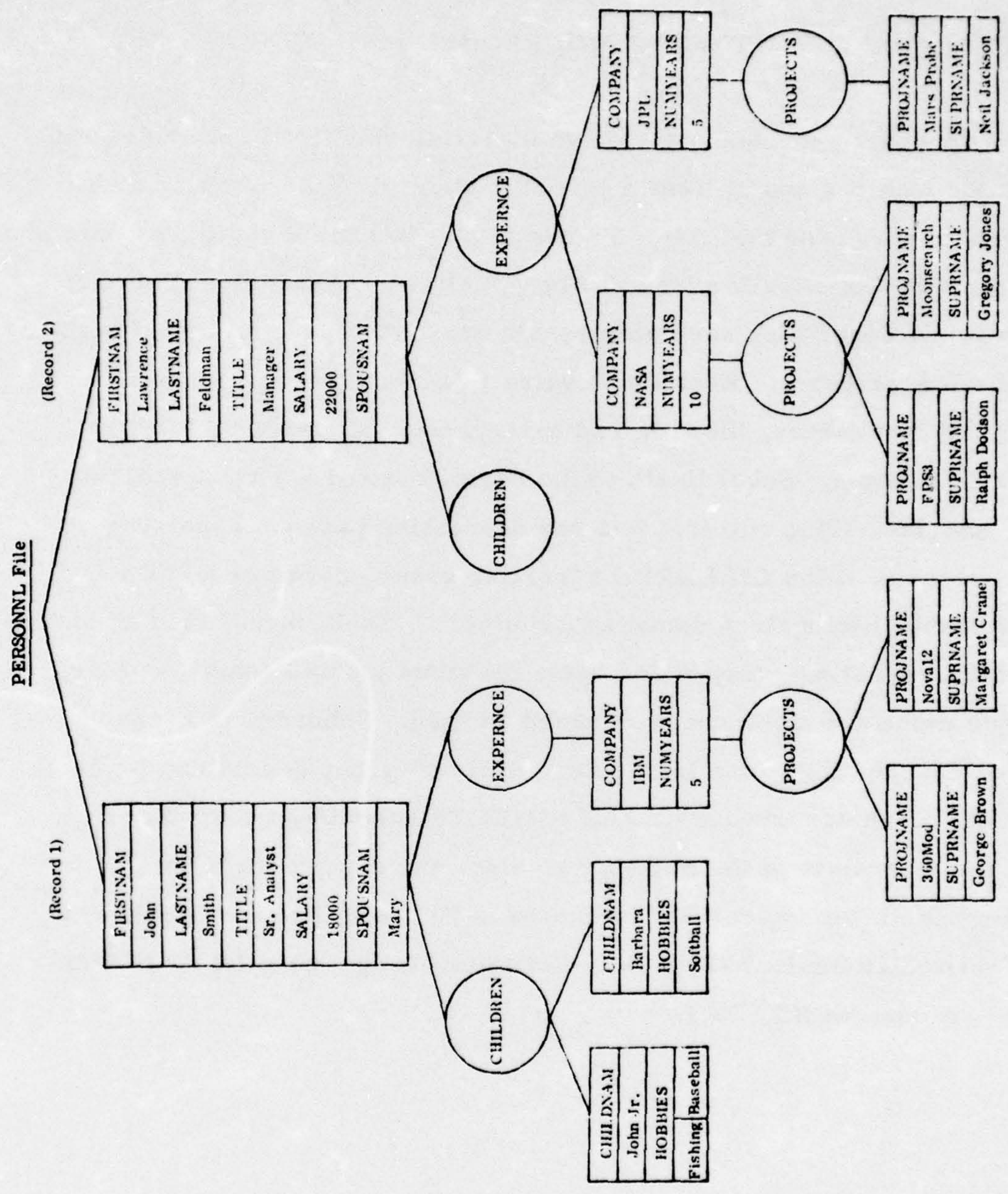
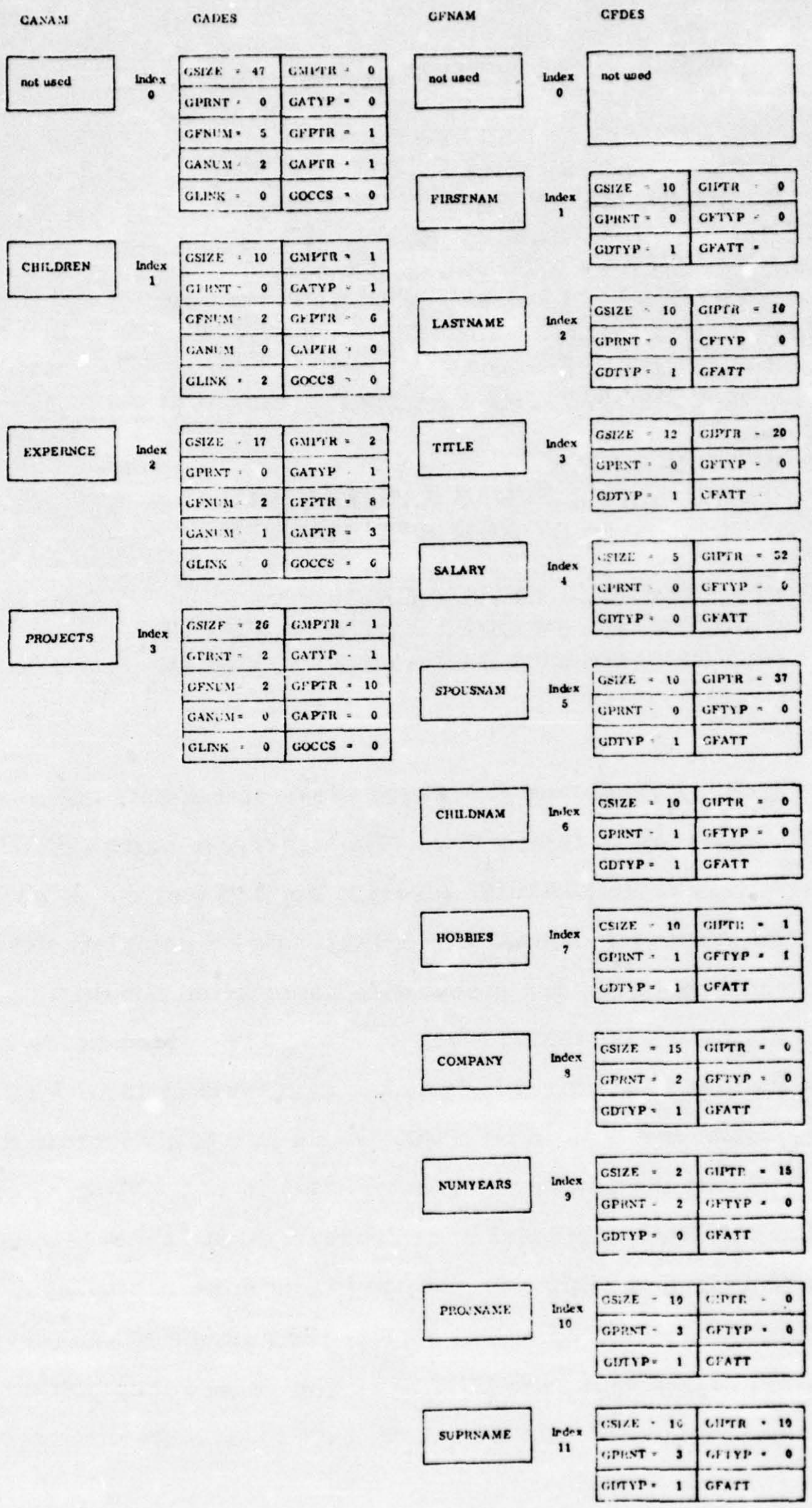


Figure B-1. UDL File.

The input for the PERSONNL file is as follows:

```
SCHEMA PERSONNL DATABASE(QLP)
FIELD FIRSTNAM SV CHAR ATT(KEY, VIS) 10
FIELD LASTNAME SV CHAR ATT(KEY, VIS) 10
FIELD TITLE SV CHAR ATT(KEY, VIS) 12
FIELD SALARY SV NUM ATT(KEY, VIS) 5
FIELD SPOUSNAM SV CHAR ATT(KEY, VIS) 10
RGROUP CHILDREN
FIELD CHILDNAM SV CHAR ATT(KEY, VIS) 10
FIELD HOBBIES MV CHAR ATT(KEY, VIS) 10
ERGROUP
RGROUP EXPERNCE
FIELD COMPANY SV CHAR ATT(KEY, VIS) 15
FIELD NUMYEARS SV NUM ATT(KEY, VIS) 2
ERGROUP
RGROUP PROJECTS EXPERNCE
FIELD PROJNAME SV CHAR ATT(KEY, VIS) 10
FIELD SUPRNAME SV CHAR ATT(KEY, VIS) 16
ERGROUP
ESCHEMA
```

The DDL process reads and processes these statements and builds the dictionary files as shown in figure B-2. The aggregate names (GANAM) and aggregate descriptions (GADES) describe the BDS and the three repeating groups. The names are stored in GANAM. For a complete description of the data stored in GADES, the global data description should be read. The GSIZE field contains the combined sizes of all single-valued fields in the aggregate. A pointer to the first field in the aggregate is in GFPTR and the number of sequential fields is in GFNUM. A pointer to a subordinate aggregate is in GAPTR and the number of subordinates is in GANUM. For aggregates other than the BDS, a pointer to its superordinate aggregate is in GPRNT and a pointer to a sibling aggregate (same superordinate) is in GLINK. For an array, GOCCS would contain the number of occurrences allowed. The aggregate type is in GATYP. The record map pointer in GMPTR is a relative pointer into GRMAP. For a complete discussion of



CANAM		CADES	
not used	Index 0	GSIZE = 47	GMPTR = 0
		GPRNT = 0	GATYP = 0
		GFNUM = 5	GFPTR = 1
		GANUM = 2	GAPTR = 1
		GLINK = 0	GOCCS = 0
CHILDREN	Index 1	GSIZE = 10	GMPTR = 1
		GPRNT = 0	GATYP = 1
		GFNUM = 2	GFPTR = 6
		GANUM 0	GAPTR = 0
		GLINK = 2	GOCCS = 0
EXPERNCE	Index 2	GSIZE = 17	GMPTR = 2
		GPRNT = 0	GATYP 1
		GFNUM = 2	GFPTR = 8
		GANUM = 1	GAPTR = 3
		GLINK 0	GOCCS = 6
PROJECTS	Index 3	GSIZF = 26	GMPTR = 1
		GPRNT = 2	GATYP = 1
		GFNUM 2	GFPTR = 10
		GANUM = 0	GAPTR = 0
		GLINK = 0	GOCCS = 0

GFNAM		GFDES	
not used	Index 0	not used	
FIRSTNAM	Index 1	GSIZE = 10	GIPTR = 0
		GPRNT = 0	GFTYP = 0
		GDTYP = 1	GFATT =
LASTNAME	Index 2	GSIZE = 10	GIPTR = 10
		GPRNT = 0	CFTYP 0
		GDTYP = 1	GFATT
TITLE	Index 3	GSIZE = 12	GIPTR = 20
		GPRNT = 0	GFTYP = 0
		GDTYP = 1	CFATT
SALARY	Index 4	GSIZE = 5	GIPTR = 32
		GPRNT 0	GFTYP = 0
		GDTYP = 0	GFATT
SPOUSNAM	Index 5	GSIZE = 10	GIPTR = 37
		GPRNT 0	GFTYP = 0
		GDTYP = 1	GFATT
CHILDNAM	Index 6	GSIZE = 10	GIPTR = 0
		GPRNT = 1	GFTYP = 0
		GDTYP = 1	GFATT
HOBBIES	Index 7	CSIZE = 10	GIPTR = 1
		GPRNT = 1	GFTYP = 1
		GDTYP = 1	GFATT
COMPANY	Index 8	GSIZE = 15	GIPTR = 0
		GPRNT = 2	GFTYP = 0
		GDTYP = 1	GFATT
NUMYEARS	Index 9	GSIZE = 2	GIPTR = 15
		GPRNT = 2	GFTYP = 0
		GDTYP = 0	GFATT
PROJNAME	Index 10	GSIZE = 10	GIPTR = 0
		GPRNT = 3	GFTYP = 0
		GDTYP = 1	CFATT
SUPRNAME	Index 11	GSIZE = 10	GIPTR = 10
		GPRNT = 3	GFTYP = 0
		GDTYP = 1	GFATT

Figure B-2. Dictionary Data for PERSONNL File.

GRMAP and its two types of nodes, global data should be consulted. The data set node for the basic data set starts, by convention, in index zero of GRMAP. Each aggregate subordinate to the BDS, then, has a GMPTR value which is relative to index zero. Figure B-3 shows actual internal records which would be built using the data shown in figure B-1 and the file dictionary data shown in figure B-2. For example, repeating group EXPERNCE has a GMPTR value of 2. Therefore, word index 2 of GRMAP contains a pointer to an occurrence node for EXPERNCE. In the case of Record 1, this pointer is 12 and at index 12 is found an occurrence node showing that EXPERNCE has one occurrence and that that occurrence is at index 14. At index 14 is a data set node for EXPERNCE showing that data for the single-valued fields start at index 98 in GRDAT. Subordinate to EXPERNCE is repeating group PROJECTS with a GMPTR value of 1. This is relative to the base of a data set node of EXPERNCE, in this case index 14. Therefore, in word index 15 is found a pointer to an occurrence node for PROJECTS.

DDL stores field names in GFNAM and field descriptive data in GFDES. Read the global data descriptions for a complete description of GFDES. The GSIZE field contains the maximum number of characters for a fixed size field and represents the actual amount of space saved in GRDAT for that field's data. For a multivalued or variable length field, GIPTR has the same use as GMPTR in GADES. It is a relative pointer into GRMAP, in this case to two words containing a pointer into GRDAT and either a character count or an occurrence count. For example, repeating group CHILDREN has a multivalued field HOBBIES. CHILDREN is subordinate to the BDS and has a GMPTR value of 1. At index 1 (for Record 1) is a pointer to an occurrence node at index 3. The occurrence node indicates two occurrences, at indexes 6 and 9. At index 6 is a data set node for CHILDREN. Field HOBBIES has a GIPTR of 1; therefore, at index 7 is a pointer into GRDAT (index 68) and at index 8 is the occurrence count.

Record 1

GRMAP

Index	
0	(1)
1	3
2	12
3	2
4	6
5	9
6	(48)
7	(68)
8	2
9	(58)
10	(88)
11	1
12	1
13	14
14	(98)
15	16
16	2
17	19
18	20
19	(115)
20	(141)

GRDAT

Index	0	1	2	3	4	5	6	7	8	9
0	–	J	O	H	N	△	△	△	△	△
10	△	S	M	I	T	H	△	△	△	△
20	△	S	R	.	△	A	N	A	L	Y
30	S	T	△	1	8	0	0	0	M	A
40	R	Y	△	△	△	△	△	△	J	O
50	H	N	△	J	R	.	△	△	B	A
60	R	B	A	R	A	△	△	△	F	I
70	S	H	I	N	G	△	△	△	B	A
80	S	E	B	A	L	L	△	△	S	O
90	F	T	B	A	L	L	△	△	I	B
100	M	△	△	△	△	△	△	△	△	△
110	△	△	△	5	△	3	6	0	M	O
120	D	△	△	△	△	G	E	O	R	G
130	E	△	B	R	O	W	N	△	△	△
140	△	N	O	V	A	1	2	△	△	△
150	△	M	A	R	G	A	R	E	T	△
160	C	R	A	N	E	△	△			

Record 2

GRMAP

Index	
0	(1)
1	0
2	3
3	2
4	6
5	13
6	(48)
7	8
8	2
9	11
10	12
11	(65)
12	(91)
13	(117)
14	15
15	1
16	17
17	(134)

GRDAT

Index	0	1	2	3	4	5	6	7	8	9
0	–	L	A	W	R	E	N	C	E	△
10	△	F	E	L	D	M	A	N	△	△
20	△	M	A	N	A	G	E	R	△	△
30	△	△	△	2	2	0	0	0	△	△
40	△	△	△	△	△	△	△	△	N	A
50	S	A	△	△	△	△	△	△	△	△
60	△	△	△	1	0	F	P	S	△	△
70	△	△	△	△	△	R	A	L	P	H
80	△	D	O	D	S	O	N	△	△	△
90	△	M	O	O	N	S	E	A	R	C
100	H	G	R	E	G	O	R	Y	△	J
110	O	N	E	S	△	△	△	J	P	L
120	△	△	△	△	△	△	△	△	△	△
130	△	△	5	△	M	A	R	S	△	P
140	R	O	B	E	N	E	I	L	△	J
150	A	C	K	S	O	N	△	△	△	△

Values in parentheses are pointers to GRDAT

Figure B-3. Record Data for PERSONNL File.

For a single-valued field, GIPTR is a relative pointer into GRDAT. For example, field LASTNAME in the BDS has a GIPTR of 10. The dataset node for the BDS starts in word zero of GRMAP and indicates that data for the BDS start in index 1 of GRDAT. Field LASTNAME's value is stored relative to 1; i. e., at index 11. Also in GFDES, GPRNT contains a pointer to the aggregate which contains the field. GFTYP contains the field's type, either single-valued, multivalued, or variable length. GDTYP specifies the field's data type, either character, numeric, or geographic. GFATT contains the interrogation and display attributes of the field.

In the two records shown in figure B-3, the record maps are relatively small. For a file which is defined with no repeating groups or arrays, the record map would consist only of the data set node for the BDS and would likely be even smaller. A large number of aggregates and nested aggregates could cause the record maps to become large.

APPENDIX C

NORMALIZATION OF BOOLEAN EXPRESSIONS

The selection-criteria portion of a UDL FIND statement must be converted to a normal form before it can be transformed to a semantically equivalent target system query statement(s). This normalization operation will be performed by the ADAPT I VALIDATE process after having first semantically validated the statement. It was decided to place this function in VALIDATE because, like VALIDATE, it also alters initial parse tree formations. Therefore, the final parse tree representation of any statement or command can be assumed after VALIDATE processing.

This appendix discusses the normalization of boolean expressions. Subjects such as disjunctive-normal-form (DNF), elementary boolean identities, and truth-table evaluation are briefly discussed. In particular, an implementation scheme is outlined for boolean expression normalization in the PDP-11 computers.

Normalization refers to two distinct processes:

a. Converting a general boolean expression to DNF.

b. Reducing the DNF to its minimal form.

Before discussing a boolean expression normalizing technique, some background material must be discussed.

Disjunctive-Normal-Form (DNF)

A boolean expression is in <u>disjunctive-normal-form</u> (DNF) when, in addition to variables, it contains no symbols other than those for conjunction (AND), disjunction (OR), and negation (NOT); negation symbols apply only to single variables; and no conjunct is a disjunction; i.e., conjunction symbols occur only between variables or their negations.

The following are examples of expressions in DNF:

(A AND B) OR (NOT A AND B)

A OR NOT B

A

A AND B

A AND B OR A (based on UDL's operator precedence).

The following examples are not in DNF:

A XOR NOT B

A NOR A

NOT (A OR B)

(A OR B) AND A

The last example is in conjunctive-normal form.

There exist three reasons for reducing general boolean expressions to DNF:

a. Removing boolean operators that may not exist in certain target systems.

b. Removing boolean parentheses that may not exist in certain target systems.

c. Providing a standard (thus predictable) expression that is manageable in separate units (disjuncts).

The first reason is directed toward the XOR and NOR operators which are not directly mappable in most target systems. The second reason is important for those systems which allow only an implicit precedence for boolean expression evaluation; i.e., left-to-right, standard precedence, etc. The third reason has stronger ramifications since it is essential for transformations involving the scope-qualifier and mixed selection-criteria; i. e., selection-criteria containing geographic rel-terms, scope-qualifiers, etc. In some systems, these exist as separate statements and therefore must be factored out.

When general boolean expressions are reduced to DNF, they usually become larger (but simpler) expressions. Therefore, they should be minimized into smaller, more compact expressions. This compact expression can be used in place of the original general boolean expression since both are logically equivalent.

Reducing a general boolean expression to DNF is conceptually a very simple process. To derive the DNF of <u>any</u> general boolean expression, proceed as follows. First, make a truth-table for the boolean expression and variables contained in it.

EXAMPLE:

Variables			Boolean Expression
A	B	C	$A \oplus \overline{(B+CA)}$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

where the foregoing symbology is interpreted as:

$-$	= negation.
$\oplus$	= exclusive-or.
$+$	= or.
juxtaposition	= and.

The foregoing example shows a boolean expression with three variables where eight value sets are possible. The boolean expression has been evaluated and its truth-value bit-string is shown below the exclusive-or operator. In order to derive the DNF for this boolean expression, all disjuncts of variables A, B, and C are required where the expression is "true" (a value of 1). In this case, the boolean expression is true for five value sets of A, B, and C. That is, the boolean expression is true when A, B, and C have the values 0 0 0, 0 0 1, 1 0 1, 1 1 0, or 1 1 1 and only for these values. When A, B, and C have the values 0 0 1, then $\overline{A}\ \overline{B}\ \overline{C}$ equals 1. Similarly, when A, B, and C equal 1 1 0, $A\ B\ \overline{C}$ equals 1. Therefore, the boolean expression can be expressed with the following five disjuncts:

$$\overline{A}\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ \overline{B}\ C + A\ B\ \overline{C} + A\ B\ C.$$

The above expression is in DNF. Note that in each disjunct a variable is complemented if its corresponding value is 0 and is not complemented if its value is 1. Also note that the exclusive-or operator has disappeared. Although the expression is logically equivalent to the original boolean expression, it can be minimized considerably.

Before describing a technique for minimizing general DNF expressions, a few initial comments are needed. The technique utilized here is a modified Quine method which requires the following elementary boolean identities:

a. $A\ (B + C) = AB + AC$

b. $A + \overline{A} = 1$

c. $A\ 1 = A$

These identities are used in minimizing boolean expressions which are in DNF.

EXAMPLE:

Step 1. $A B \overline{C} + A B C$

Step 2. $AB (\overline{C} + C)$, identity a

Step 3. $AB (1)$, identity b

Step 4. AB, identity c

With repeated (and selective) use of these three boolean identities, the foregoing large DNF expression can be minimized to the following form:

$\overline{A}\ \overline{B} + AC + AB\overline{C}$.

The remainder of this discussion is concerned with the implementation of the foregoing material.

Boolean Expression Normalizer (BEN)

The Boolean Expression Normalizer (BEN) function operates as follows. BEN scans a parse tree for selection-criteria. If detected, it is expanded to disjunctive-normal-form. The resulting normalized boolean-expressions are then minimized and inserted into the parse tree replacing the original expressions.

BEN is composed of two main subcomponents, a truth-table evaluator and a normalizer/minimizer. For this particular implementation scheme, two assumptions are made:

a. The parse tree representing the boolean expression has the proper operator precedence built into it.

b. The boolean expression can not have more than eight unique variables (rel-terms) contained in it.

This last restriction is not that limiting to the user. A boolean expression or selection-criteria containing more than eight unique variables would not be very common and, if necessary, can be expressed using two or more smaller expressions; i.e., dependent interrogations.

TRUTH-TABLE EVALUATOR – Each rel-term must be identified and sorted from the other rel-terms. For two rel-terms to be identical (and thus representing a single boolean variable), they must specify the same rel-op, field-term, and data-constant(s). Each unique rel-term is assigned a boolean variable starting with V_0 (up to V_7).

Since up to eight variables are allowed, the truth-value set for these eight variables may be a set of bit-strings each 256 bits long (2^8). Therefore, each variable must be 16 computer words in length. Boolean variable V_0 will be represented by 16 words of alternating zeros and ones, variable V_1 will be represented by 16 words of alternating 0 0 and 1 1, and so on.

The parse tree is now evaluated in the standard fashion using the boolean variables. Two 16-word variables will also be needed to contain intermediate results as well as the final result of evaluation. Completion of the parse tree evaluation provides a 16-word bit-string (less than this if fewer than eight variables are involved) representing the truth-value of the boolean expression. This bit-string is used by the next phase of BEN, the normalizer/minimizer.

NORMALIZER/MINIMIZER – Before discussing the operation of this phase of BEN, a description of the MINTERM table is in order (see figure C-1). Depending on the number of rel-terms (or variables), space is allocated in core for the table (and freed after the minimization is done).

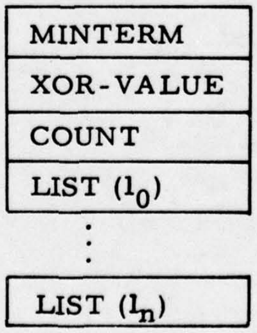


Figure C-1. Pictorial Representation of a MINTERM Item.

This table has 2^n minterms for n variables. A minterm of n variables is a boolean product of these n variables, with each variable present in either its complemented or uncomplemented form. The right-most bit of the MINTERM represents V_0, the next bit V_1, and so on for n variables. For example:

0000	$\overline{V}_3 \overline{V}_2 \overline{V}_1 \overline{V}_0$
0100	$\overline{V}_3 V_2 \overline{V}_1 \overline{V}_0$
1011	$V_3 \overline{V}_2 V_1 V_0$

The first minterm has the value 0, the second has the value 1, and the last has the value $(2^n - 1)$.

Initially, the XOR-VALUE fields are set to all ones (377_8). In order to establish the DNF of a boolean expression, the truth-value bit-string generated by the Truth-Table-Evaluator is analyzed bit by bit. For each bit set, the corresponding XOR-VALUE field is cleared (set to zero). The first bit corresponds to the first minterm, the second bit to the second minterm, and so on. The minimization pass utilizes the MINTERM table directly once the applicable minterms have had their XOR-VALUEs set to zero.

The XOR-VALUE field is altered during the minimization process. When two minterms are simplified, one of the minterm's XOR-VALUE reflects that a variable has been effectively factored out. A "1" is placed in the XOR-VALUE in that variable's position.

XOR-VALUE	MINTERM	Represented Disjunct
0000	$V_3 V_2 \overline{V}_1 V_0$	$V_3 V_2 \overline{V}_1 V_0$
0100	$V_3 V_2 \overline{V}_1 V_0$	$V_3 \overline{V}_1 V_0$

XOR-VALUE	MINTERM	Represented Disjunct
0101	$V_3\ V_2\ \overline{V}_1\ V_0$	$V_3\ \overline{V}_1$
1111	$V_3\ V_2\ \overline{V}_1\ V_0$	–

Therefore, each minterm whose XOR-VALUE does not equal 377_8 is a disjunct of the boolean expression. The corresponding MINTERM value represents the variables, complemented or uncomplemented, of this disjunct.

The structure of the MINTERM table lends itself to the minimization technique described earlier in this appendix. The repeated application of the three boolean identities discussed is performed on the tables. The three rules can be shortened to a single boolean identity:

$$AB + A\overline{B} = A$$

The mechanics of the utilization of the identity can be briefly outlined as:

a. Two disjuncts are found that have the same MINTERMs, where one and only one variable differs with respect to whether or not it is complemented.

b. This variable is factored out (by placing a "1" in that variable's position) in one of the two XOR-VALUEs.

c. The other disjunct is deleted by setting the XOR-VALUE to 377_8.

The following text goes into greater depth on the minimization algorithm. Each minimization loop consists of a counting scheme, and a procedure for the minimization of minterms. These iterations continue until there are no more possible reductions.

A count is tabulated for each applicable minterm, MT_x (a minterm whose XOR-VALUE does not equal 377_8), denoting the number of minterms that it can be simplified with. The values of these minterms are stored in

$LIST_x$ for future reference. To determine if two minterms (MT_x and MT_y) can be simplified, two conditions must be satisfied:

a. The current XOR-VALUEs must be equal. (This assumes that the minterms have the same variables present.)

b. The value obtained from $(MT_x \oplus MT_y) \cdot \overline{\text{XOR-VALUE}}$ contains one and only one bit which is set to one (any bit).

MAXCNT equals the largest count of any minterm for a particular pass. Depending on the counts, simplification can be performed. At this time, a discussion on bit manipulation during the simplication process is in order. When two minterms are to be simplified, one of the minterm's XOR-VALUE is set to the resultant of XOR-VALUE + $MT_x \oplus MT_y$ which essentially factors out the variable that differs between MT_x and MT_y. The other XOR-VALUE is deleted (XOR-VALUE is set to 377_8) except in a special case discussed later. Both minterms' counts are set to zero. Consider the following example:

	MINTERM	XOR-VALUE	Represented Disjunct
MT_{15}	1111	1000	$V_2 V_1 V_0$
MT_5	0101	1000	$V_2 \overline{V}_1 V_0$

The XOR-VALUEs are equal. The result of

$$\begin{aligned}(MT_{15} \oplus MT_5) \cdot \overline{\text{XOR-VALUE}} &= (1111 \oplus 0101) \cdot \overline{1000} \\ &= 1010 \cdot 0111 \\ &= 0010\end{aligned}$$

has one and only one bit set (bit 1); therefore, minterms MT_{15} and MT_5 can be simplified.

$\text{XOR-VALUE}_5 = 377_8$

$\text{XOR-VALUE}_{15} = MT_{15} \oplus MT_5 + \text{XOR-VALUE}_{15}$

$= 1010 + 1000$

$= 1010$

Variable V_1 has been factored out.

The represented disjunct of MT_{15} (1 1 1 1) with an $\text{XOR-VALUE}_{15} =$ 1010 is $V_2\ V_0$.

The process of selecting which minterms should be combined is dependent on the counts of the minterms. Obviously, minterms with zero counts can not be further simplified. A minterm which can combine with only one other minterm is simplified with that minterm if it also has a count of 1.

All minterms which have counts equal to MAXCNT combine in the following way (let MT_x be such a minterm thereby implying that $COUNT_x =$ MAXCNT):

a. Minimize MT_x with those minterms in $LIST_x$ whose count equals 1. Under such conditions, the simplification process deviates from the normal procedure. The basic process is outlined below:

1. MT_z represents a minterm with a count of 1 in $LIST_x$.
2. XOR-VALUE_z is set equal to $\text{XOR-VALUE}_z + MT_x \oplus MT_z$.
3. $\text{Count}_z = 0$.
4. Count_x is decremented by 1.

b. Choose a minterm MT_y whose count is the largest of those minterms in $LIST_x$.

c. Minimize MT_y with those minterms in $LIST_y$ whose count equals 1. (Refer to step a.)

d. If after steps a and c $COUNT_y$ and $COUNT_x$ are equal to 1, MT_y and MT_x are not minimized with each other. (Their minimization would be redundant.) Otherwise, minimize MT_x and MT_y using the normal procedures.

EXAMPLE 1:

Assume four variables are present (V_0, V_1, V_2, and V_3). The five miniterms whose XOR-VALUEs are not set to 377_8 are minimized below.

	V_3	V_2	V_1	V_0
MT_0	0	0	0	0
MT_1	0	0	0	1
MT_8	1	0	0	0
MT_9	1	0	0	1
MT_{12}	1	1	0	0

Pass 1 – MT_8 Combined with MT_{12} and MT_0.

	XOR-VALUE	COUNT	LIST
MT_0	0000	2	1, 8
MT_1	0000	2	0, 9
MT_8	0000	3	0, 9, 12
MT_9	0000	2	1, 8
MT_{12}	0000	1	8

Pass 2 – MT_1 Combined with MT_9.

	XOR-VALUE	COUNT	LIST
MT_0	1111		
MT_1	0000	1	9
MT_8	1000	0	-
MT_9	0000	1	1
MT_{12}	0100	0	-

Pass 3 – MT_1 Combined with MT_8

	XOR-VALUE	COUNT	LIST
MT_0	1111		
MT_1	1000	1	8
MT_8	1000	1	1
MT_9	1111		
MT_{12}	0100	0	-

Pass 4 – MAXCNT = 0

	XOR-VALUE	COUNT	LIST
MT_0	1111		
MT_1	1001	0	-
MT_8	1111		
MT_9	1111		
MT_{12}	0100	0	-

Final reduction: $\overline{V_2 V_1} + V_3\overline{V_1 V_0}$.

EXAMPLE 2:

	V_3	V_2	V_1	V_0
MT_0	0	0	0	0
MT_2	0	0	1	0
MT_4	0	1	0	0
MT_7	0	1	1	1
MT_{11}	1	0	1	1
MT_{12}	1	1	0	0
MT_{13}	1	1	0	1
MT_{15}	1	1	1	1

Pass 1 – MT_{15} Combined with MT_7, MT_{11}, MT_{13}

	XOR-VALUE	COUNT	LIST
MT_0	0000	2	2,4
MT_2	0000	1	0
MT_4	0000	2	0,12
MT_7	0000	1	15
MT_{11}	0000	1	15
MT_{12}	0000	2	4,13
MT_{13}	0000	2	12, 15
MT_{15}	0000	3	7, 11, 13

Pass 2 – MT_0 Combined with MT_2

	XOR-VALUE	COUNT	LIST
MT_0	0000	2	2,4
MT_2	0000	1	0
MT_4	0000	2	0,12
MT_7	1000	0	-
MT_{11}	0100	0	-
MT_{12}	0000	1	4
MT_{13}	1111		
MT_{15}	0010	0	-

Pass 3 – MAXCNT = 0

	XOR-VALUE	COUNT	LIST
MT_0	1111	0	-
MT_2	0010	0	-
MT_4	1111		
MT_7	1000	0	-
MT_{11}	0100	0	-
MT_{12}	1000	0	-
MT_{13}	1111		
MT_{15}	0010	0	-

Final reduction: $\overline{V_3 V_2 V_0} + V_2 V_1 V_0 + V_3 V_1 V_0 + V_2 \overline{V_1 V_0} + V_3 V_2 V_0$.

Upon completion of the foregoing steps, a new count is made. If MAXCNT equals zero, no further reductions can be made; otherwise, the minimization process continues. See flow chart of this algorithm in figure C-2.

The reason for calculating a simplification count prior to actual minimization is as follows. Intuitively, one would think it doesn't really matter which minterms are combined during the minimization process. However, it is possible to make initial combinations which inhibit multiple combinations that may be possible in later passes, thus not producing the minimal expression. Therefore, only those minterms which combine with the most minterms are used initially.

EXAMPLE:

$$AB\overline{C} + \overline{A}\overline{B}C + ABC + A\overline{B}\overline{C} + A\overline{B}C$$

$AB\overline{C}$ combines with <u>two</u> disjuncts, ABC and $A\overline{B}\overline{C}$.

$\overline{A}\overline{B}C$ combines with <u>one</u> disjunct, $A\overline{B}C$.

ABC combines with <u>two</u> disjuncts, $AB\overline{C}$ and $A\overline{B}C$.

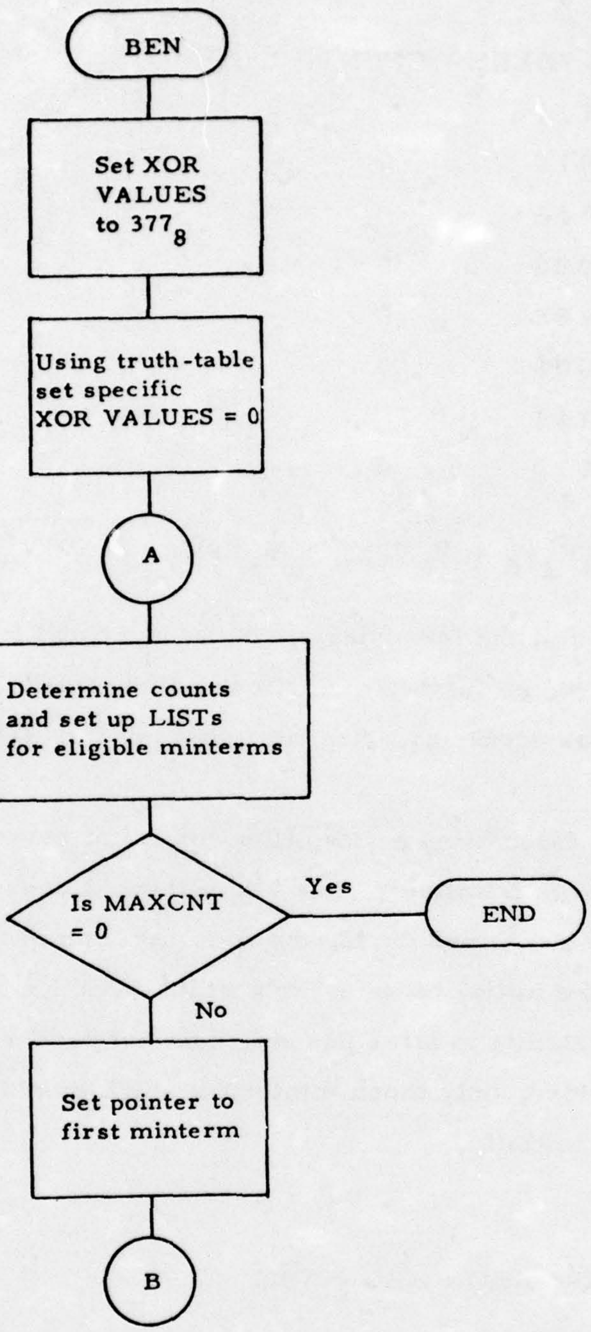

BEN
Set XOR VALUES to 377_8
Using truth-table set specific XOR VALUES = 0
A
Determine counts and set up LISTs for eligible minterms
Is MAXCNT = 0
Yes
END
No
Set pointer to first minterm
B

Figure C-2. Minimization Algorithm (Sheet 1 of 2).

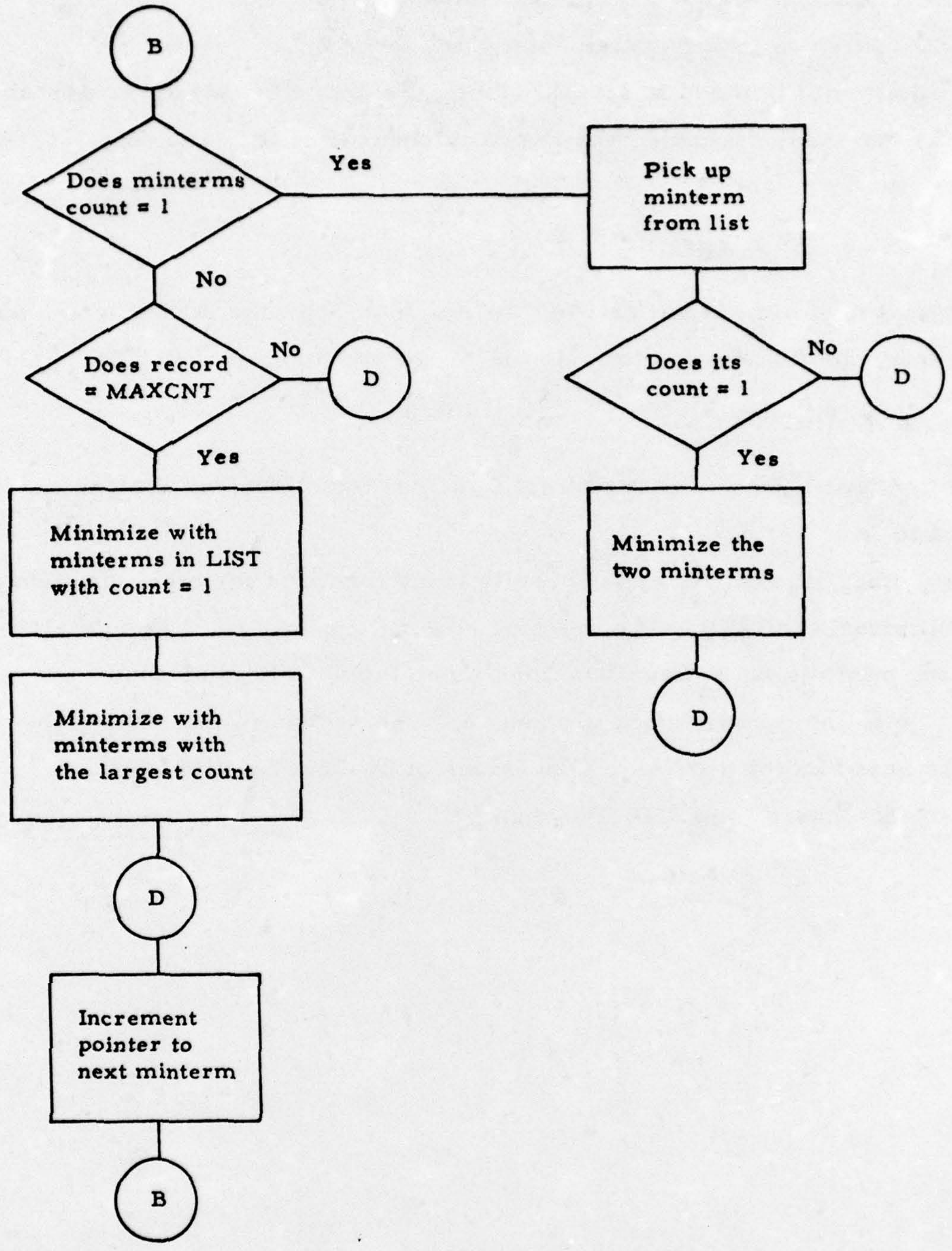

B
Does minterms count = 1
Yes
No
Pick up minterm from list
Does record = MAXCNT
No
D
Does its count = 1
No
D
Yes
Yes
Minimize with minterms in LIST with count = 1
Minimize the two minterms
D
Minimize with minterms with the largest count
D
Increment pointer to next minterm
B

Figure C-2. Minimization Algorithm (Sheet 2 of 2).

$A\overline{B}\,\overline{C}$ combines with two disjuncts, $AB\overline{C}$ and $A\overline{B}C$.

$A\overline{B}C$ combines with two disjuncts, $\overline{A}\,\overline{B}C$ and $A\overline{B}\,\overline{C}$.

If an attempt is made to simplify using disjunct $\overline{A}\,\overline{B}C$ (which combines with only one other disjunct), the result is reduced to the following expression:

$$\overline{B}C + A\overline{C} + ABC$$

However, if simplification uses the disjuncts with the largest combination counts, the following expression is produced:

$$A + \overline{A}\,\overline{B}C$$

Notice that $\overline{A}\,\overline{B}C$ was never simplified, but the other four disjuncts reduced to A.

Two other algorithms are currently being explored for their efficiency and minimization ability. The more promising one is J. C. Wilson's algorithm for minimizing boolean functions using the Cranfield method as a first stage in the minimization procedure. The second minimizing algorithm is based on the ring-sum expansions of boolean functions by G. Bioul, M. Davio, and J. P. Deschamps.

DISTRIBUTION LIST

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, VA 22217	2 copies
Office of Naval Research Code 102IP Arlington, VA 22217	6 copies
Office of Naval Research Code 200 Arlington, VA 22217	1 copy
Office of Naval Research Code 455 Arlington, VA 22217	1 copy
Office of Naval Research Code 458 Arlington, VA 22217	1 copy
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, MA 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, IL 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA 91106	1 copy

New York Area Office	1 copy
715 Broadway – 5th Floor	
New York, NY 10003	
Naval Research Laboratory	6 copies
Technical Information Division, Code 2627	
Washington, D. C. 20375	
Dr. A. L. Slafkosky	1 copy
Scientific Advisor	
Commandant of the Marine Corps (Code RD-1)	
Washington, D. C. 20380	
Naval Electronics Laboratory Center	1 copy
Advanced Software Technology Division	
Code 5200	
San Diego, CA 92152	
Mr. E. H. Gleissner	1 copy
Naval Ship Research & Development Center	
Computation and Mathematics Department	
Bethesda, MD 20084	
Captain Grace M. Hopper	1 copy
NAICOM/MIS Planning Branch (OP-916D)	
Office of Chief of Naval Operations	
Washington, D. C. 20350	
Mr. Kin B. Thompson	1 copy
Technical Director	
Information Systems Division (OP-91T)	
Office of Chief of Naval Operations	
Washington, D. C. 20350	
Advanced Research Projects Agency	1 copy
Information Processing Techniques	
1400 Wilson Boulevard	
Arlington, VA 22209	